

移动开发丛书

从基本语法
到应用开发
拒绝纸上谈兵
电商App 开发实录
尽现Kotlin魅力

Kotlin从零到精通
Android开发

•欧阳燊 编著•

Android老司机带你玩转Kotlin

清华大学出版社



移动开发丛书

Kotlin从零到精通

Android开发

•欧阳燊 编著•

清華大学出版社
北 京

内 容 简 介

本书是一部讲解Kotlin语言的入门书籍，从Kotlin语言的基本语法一直讲到如何将其运用于Android开发。由浅入深、从理论到实战，帮助读者快速掌握Kotlin开发技巧。

全书共有10章内容，可分为三大部分：第一部分即第1章，主要介绍Kotlin语言的开发环境搭建；第二部分包含第2~5章，主要介绍Kotlin的基本语法知识，包括Kotlin的变量声明、控制语句、函数定义、类与对象等；第三部分包含第6~10章，主要介绍如何使用Kotlin进行实际的App开发工作，包括利用Kotlin操作简单控件、复杂控件、数据存储、自定义控件、网络通信等。为增强学习Kotlin语言的趣味，本书在讲解Kotlin的用法时，特别注意结合生活中的具体案例，并加以示范和运用。尤其是后面讲到利用Kotlin开发App的时候，精心设计了数个电商App的实战模块，例如电商App的登录模块、频道模块、购物车模块、团购模块、升级模块等。通过这些实战小项目，读者可迅速将Kotlin应用于App开发工作中。

本书适用于Android开发的广大从业者、Kotlin语言的业余爱好者，也可用作大中专院校与培训机构的Kotlin课程教材。

图书在版编目（CIP）数据

Kotlin从零到精通Android开发/欧阳燊编著.—北京：清华大学出版社，2018
（移动开发丛书）
ISBN 978-7-302-49814-8

Ⅰ. ①K…　Ⅱ. ①欧…　Ⅲ. ①JAVA语言—程序设计　Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2018）第037614号

责任编辑：王金柱
封面设计：王　翔
责任校对：闫秀华
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**21.75　　**字　　数：**557千字
版　　次：2018年4月第1版　　**印　　次：**2018年4月第1次印刷
印　　数：1~3500
定　　价：79.00元

产品编号：077880-01

前　　言

新技术的发展日新月异，编程语言也不例外，从早期的机器语言到汇编语言，再到以 C 语言为代表的高级语言，一路衍生了 C++、Java、Objective-C 等庞大的编程语言家族。其中，Java 经过多年的发展已经是一枝独秀，不但在服务端的开发中占据优势，而且在客户端的安卓开发上也形成垄断之势。不过，由于 Java 语言诞生较早（诞生于 20 世纪 90 年代中期），使得它不可避免地存在一些先天不足，比如业务代码过于冗长、处理逻辑不够灵活、安全隐患层出不穷等。鉴于此，一方面 Java 语言不断更新换代，到 2017 年已经迭代到了 Java 9 版本；另一方面，人们也试图设计新的语言以便更好地“填坑”，于是涌现了 Scala、Groovy、Clojure 等新兴语言，而 Kotlin 就是这些新兴语言中的佼佼者。

Kotlin 问世于 2011 年，作为后起之秀的它虽然拥有代码简洁、函数式编程、更安全健壮、百分百兼容 Java 等诸多特性，但是前有 C++、Java 等老语言根深叶茂，后有 Python、Go 等新语言紧追不舍，Kotlin 头几年的发展一直不温不火。直到这两年，在 JetBrains、Google 等公司的大力扶持之下，Kotlin 的发展才驶上了快车道，先是在 2016 年 2 月推出 Kotlin 1.0 发布版，再是谷歌公司在 2017 年 5 月宣布将 Kotlin 作为 Android 的官方开发语言，然后在 2017 年 10 月推出的 Android Studio 3.0 正式集成了 Kotlin 开发环境，紧接着更完善的 Kotlin 1.2 正式版在 2017 年 11 月发布。正如当初 Android Studio 取代 Eclipse 成为 Android 开发的主流开发工具一样，在可预见的未来，Kotlin 必将逐步取代 Java 成为主流的 App 开发语言。

被寄予厚望的 Kotlin 在编程工作中给开发者带来的巨大便利毋庸置疑，大量的开发实践表明，实现同样功能的 Kotlin 代码往往只有对应 Java 代码的三分之一。并且 Kotlin 的语法兼容并蓄、易懂易学，只要开发者拥有任何一门高级语言的编程基础，再配合一本合适的 Kotlin 入门教程，短时间内即可成为 Kotlin 熟练开发者。正因为 Kotlin 是如此的简单易用，它的代码也是如此的简洁明了，所以倘若介绍 Kotlin 语法的教程还在长篇大论，那它一定是在夸夸其谈地“耍流氓”。Kotlin 的设计理念是尽可能的简易，而不是抛出一堆令人生畏的烦琐概念，因此本书在介绍 Kotlin 用法的时候，也秉承了与之相符的一贯理念，即凡是能够简单处理的事情，决不拐弯抹角、拖泥带水。

本书既是一本 Kotlin 语法的入门教程，也是讲述 Kotlin 开发 App 的教程，一方面介绍 Kotlin 语言的基本语法，另一方面介绍 Kotlin 在安卓开发中的实际应用，可谓是结合理论、联系实战，方便读者迅速将 Kotlin 运用于日常的 App 开发工作之中，更好、更快地将学习成果展现出来，起到立竿见影的工作成效。当然，本书的侧重点在于教会读者利用 Kotlin 进行安卓开发工作，故而在有限的内容篇幅上有所取舍，比如服务端的 Kotlin 开发着墨不多，另外阐述了如何使用 Kotlin 实现常见的几种 App 开发技术，其余的 App 开发涉及的 Kotlin 技术即可触类旁通。如果读者想要了解更详细、更具体的 App 开发技能，可参见笔者的另一部 App 开发专著《Android Studio 开发实战：从零基础到 App 上线》。

全书共有 10 章内容，循序渐进，可分为三大部分：第一部分即第 1 章，主要介绍 Kotlin 语言的开发环境搭建；第二部分包含第 2~5 章，主要介绍 Kotlin 的基本语法知识，包括 Kotlin 的变量声明、控制语句、函数定义、类与对象等；第三部分包含第 6~10 章，主要介绍如何使用 Kotlin 进行实际的 App 开发工作，包括如何通过 Kotlin 使用简单控件、如何通过 Kotlin 操纵复杂控件、如何通过 Kotlin 进行数据存储、如何通过 Kotlin 自定义控件、如何通过 Kotlin 实现网络通信等。通过本书这 10 章的学习，读者应该能够掌握 Kotlin 的大部分常用语法，并将其得心应手地运用于 App 开发工作中，你会发现多了一门可供选择的 App 开发语言是多么奇妙的事情。

从零开始学 Kotlin 其实指的是 Kotlin 零基础，并非编程零基础。在学习本书之前，读者应当掌握至少一门高级开发语言。如果没有任何编程基础就来学习 Kotlin，这是不现实也是不可取的，因为短期之内各公司不会招聘只会 Kotlin 的程序员，而且 Kotlin 在 Android 开发中取代 Java 也必然是个缓慢的进程。所以学习 Kotlin 不提倡急于求成，但这并不意味着 App 开发者可以对 Kotlin 熟视无睹，任何一个新事物都有其发展壮大的过程，同时机会都是留给有准备的人。与其等 Kotlin 形成燎原之势才后知后觉地学习它，不如现在就未雨绸缪地掌握它，技术投资得越早，未来的开发收益就越大。

本书的所有代码例子都基于 Android Studio 3.0 和 Kotlin 1.2 开发，并使用 API 27 的 SDK（Android 8.1）编译与调试通过。所有的附录源代码均可在网络上下载，具体下载方式可访问笔者的博客 http://blog.csdn.net/aqi00。读者也可以从以下地址下载本书源代码：

https://pan.baidu.com/s/1ceRZzDK4_zT-uQHqy2WFHw（注意区分数字和英文字母大小写）

如果下载有问题，请发送电子邮件至 booksaga@126.com，邮件标题为“Kotlin 从零到精通 Android 开发配书源代码”获得帮助。

读者在阅读本书时，若对书中内容有疑问，也可在该博客上留言。或者关注笔者的微信公众号“老欧说安卓”，更快更方便地阅读技术干货。

最后感谢王金柱编辑以及各位出版社同仁的热情指点和密切配合，感谢我的家人一直以来的支持，如果没有大家的鼎力协助，就没有本书的顺利完成。

欧阳燊

2018 年 1 月

目　　录

第 1 章

搭建 Kotlin 开发环境

本章主要介绍 Kotlin 开发环境的搭建过程，首先阐述 Kotlin 语言与 Android 开发之间的关系，接着描述 Kotlin 开发工具，也就是 Android Studio 的安装和启动步骤，然后说明 SDK 及其相关插件的安装与升级方法，接着论述如何对 Kotlin 工程的编译配置进行调整，最后演示 Kotlin 新技术带来哪些革命性的变化。

1.1 Kotlin 与 Android 开发的关系

本节主要介绍 Kotlin 语言与 Android 开发之间的关系，包括 Kotlin 的基本概念及其特殊优势，以及 Kotlin 被谷歌钦定为 Android Studio 官方开发语言之后的发展大事。

1.1.1 Kotlin 语言简介

Kotlin 是一种基于 JVM 的新型编程语言，它完全兼容 Java 语言，Kotlin 代码可以编译成 Java 字节码，也可以编译成 JavaScript，方便在没有 JVM 的设备上运行。与流行的 Java 语言比较，Kotlin 具备下列优势：

（1）Kotlin 更简洁，完成同样的业务功能，Kotlin 代码通常只有对应 Java 代码的三分之一。

（2）Kotlin 更安全，它能够在编码阶段就自动检测常见的 BUG，比如引用了空指针等。

（3）Kotlin 更强大，它提供了扩展函数、默认参数、接口委托、属性代理等 Java 所不具备的高级特性，从而可以完成更复杂的业务逻辑。

1.1.2 Android Studio 的官方开发语言

Kotlin 很早就被运用到 Android 开发之中，之前一直作为 Android Studio 的插件提供下载，Android Studio 只要安装了 Kotlin 插件，就能用来开发 Kotlin 编码的 App 工程。

2017 年 5 月，谷歌宣布将 Kotlin 纳入 Android Studio 开发的官方语言，这意味着 Android Studio 对 Kotlin 的编译支持会大大增强，由此掀起了广大安卓开发者学习 Kotlin 编程的热潮。

2017 年 10 月，Android Studio 推出 3.0 正式版，从 3.0 版本开始，Android Studio 自动集成 Kotlin 插件，在安装 Android Studio 3.0 时就连带配置了 Kotlin 的开发环境。

2017 年 11 月，Kotlin 语言推出 1.2 发布版，该版本的 Kotlin 具备更好的跨平台特性，编译性能也比 1.1 版提高了 25%左右，同时也更好地支持 Android 开发。

1.2 Kotlin 开发工具

本节主要介绍 Kotlin 开发环境的搭建以及 Kotlin 工程的基本操作，包括安装 Android Studio 的具体步骤、启动 Android Studio 的详细配置、如何创建一个 Kotlin 工程、如何新建各种 Kotlin 文件等。

1.2.1 安装 Android Studio

Android Studio 的官方下载页面是 https://developer.android.google.cn/studio/index.html，在这里可以找到 Android Studio 的下载地址与使用教程。首先把最新版本的 Android Studio 下载到电脑本地，然后双击下载完成的 Android Studio 安装程序，弹出安装欢迎对话框，如图 1-1 所示。单击该对话框右下方的“Next”按钮，跳到下一页的许可同意对话框，单击“Agree”按钮，进入下一页的组件选择对话框，如图 1-2 所示。




图 1-1 Android Studio 的安装欢迎对话框

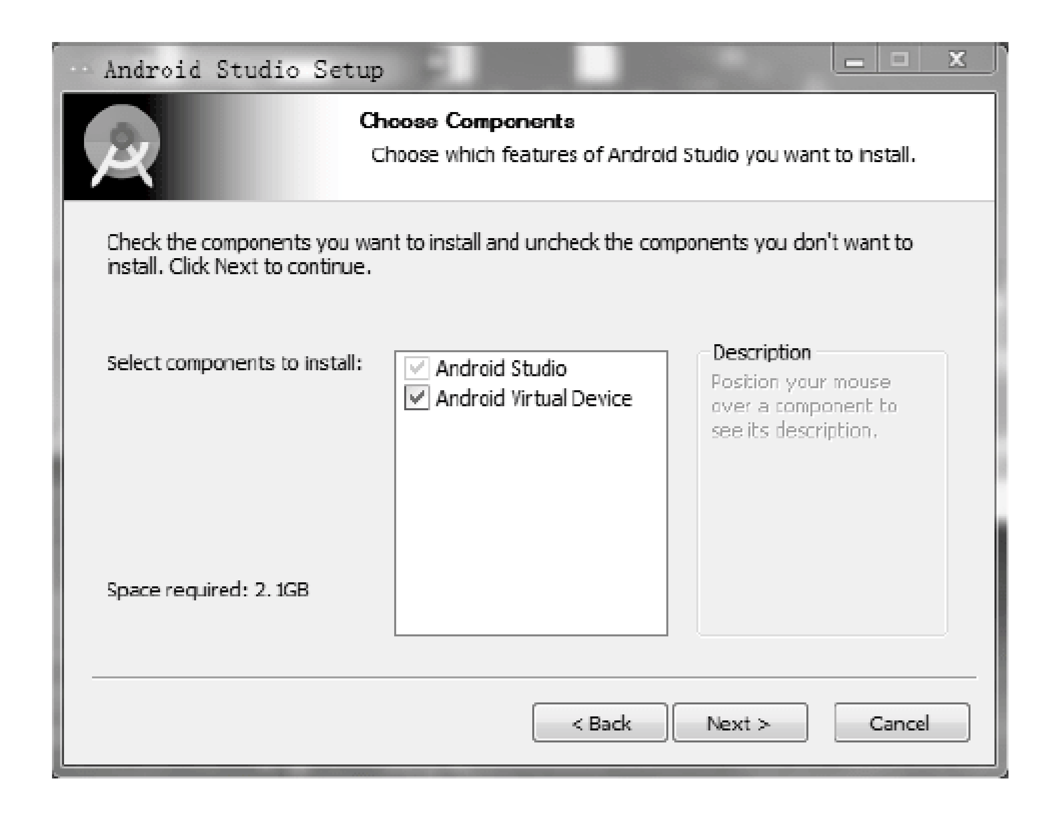


图 1-2 Android Studio 的组件选择对话框

全部勾选，然后单击“Next”按钮，跳转到安装路径配置对话框。建议将 Android Studio 安装

在除系统盘外的其他磁盘，比如 D 盘，如图 1-3 所示，然后单击“Next”按钮。在下一个对话框中选择开始菜单的目录，这里使用默认的“Android Studio”，如图 1-4 所示，然后单击“Install”按钮，等待安装过程进行。

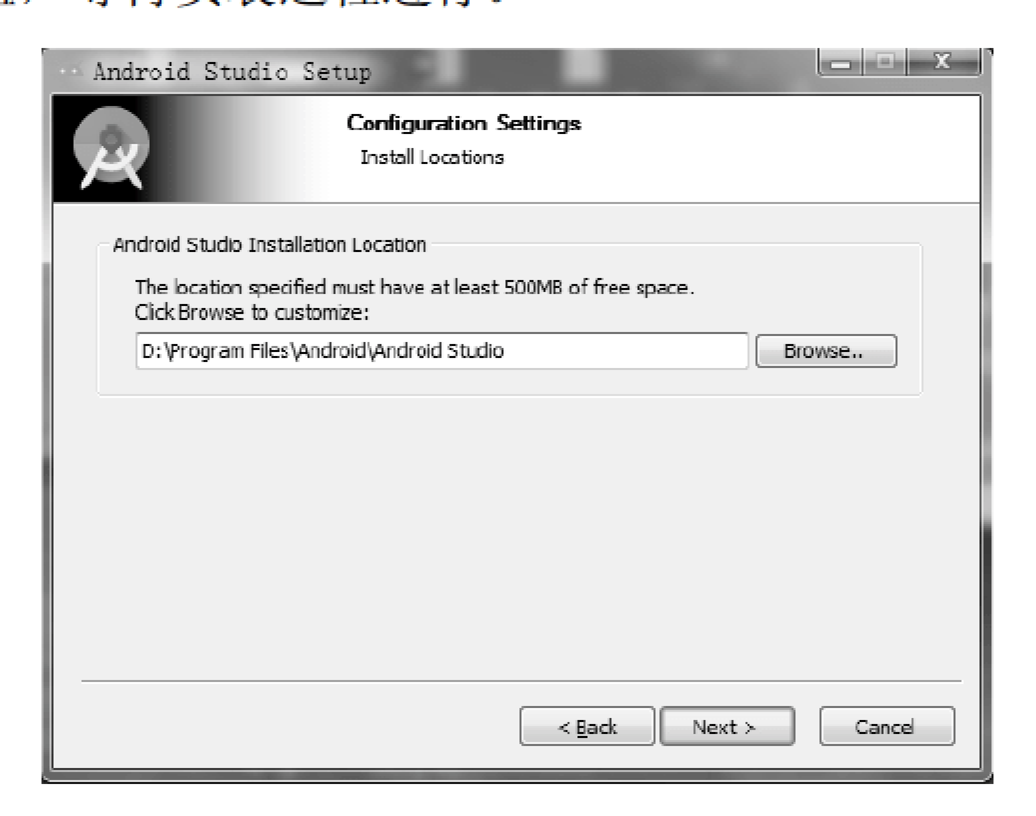


图 1-3　Android Studio 的安装目录对话框

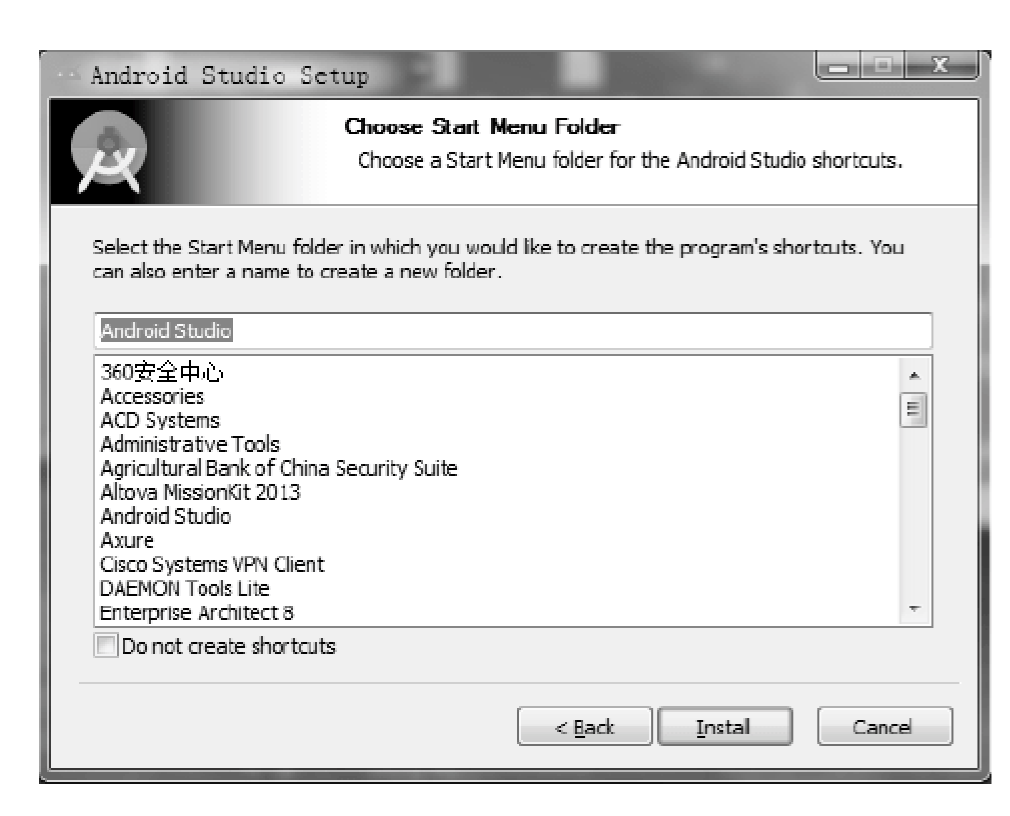


图 1-4　Android Studio 的安装设置对话框

安装过程的进度对话框如图 1-5 所示，进度完成的结果对话框如图 1-6 所示，单击“Next”按钮结束安装操作。

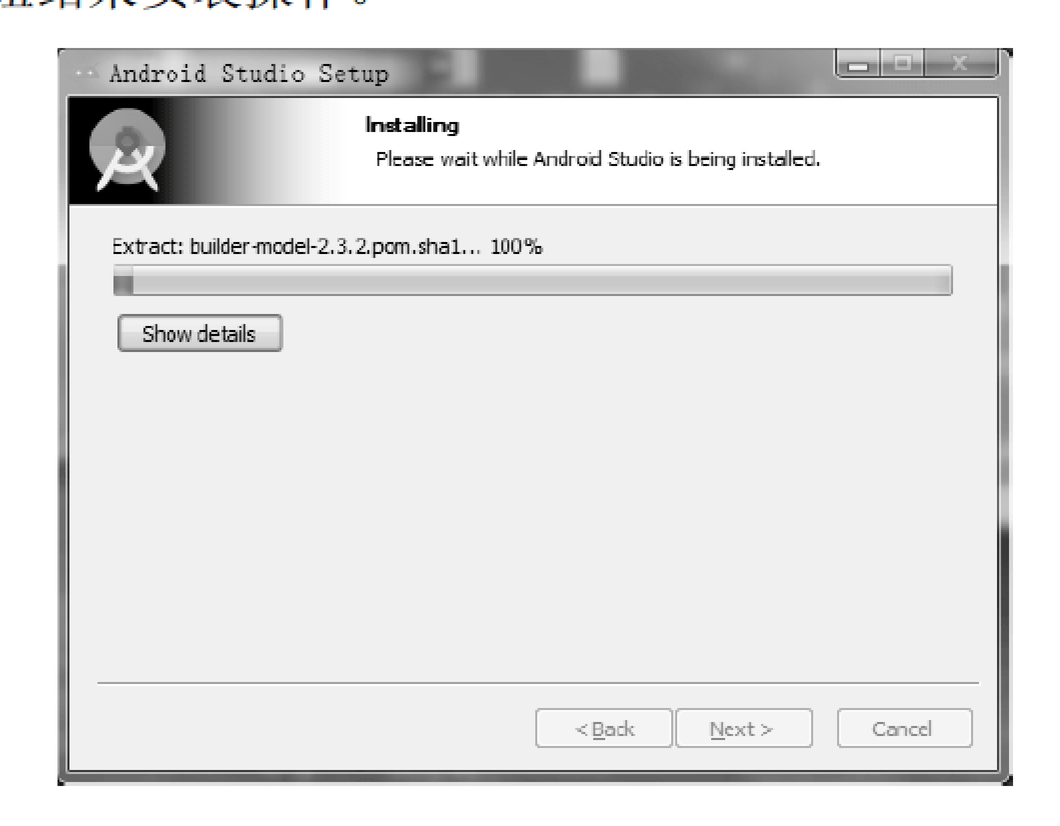


图 1-5　Android Studio 的安装进度对话框

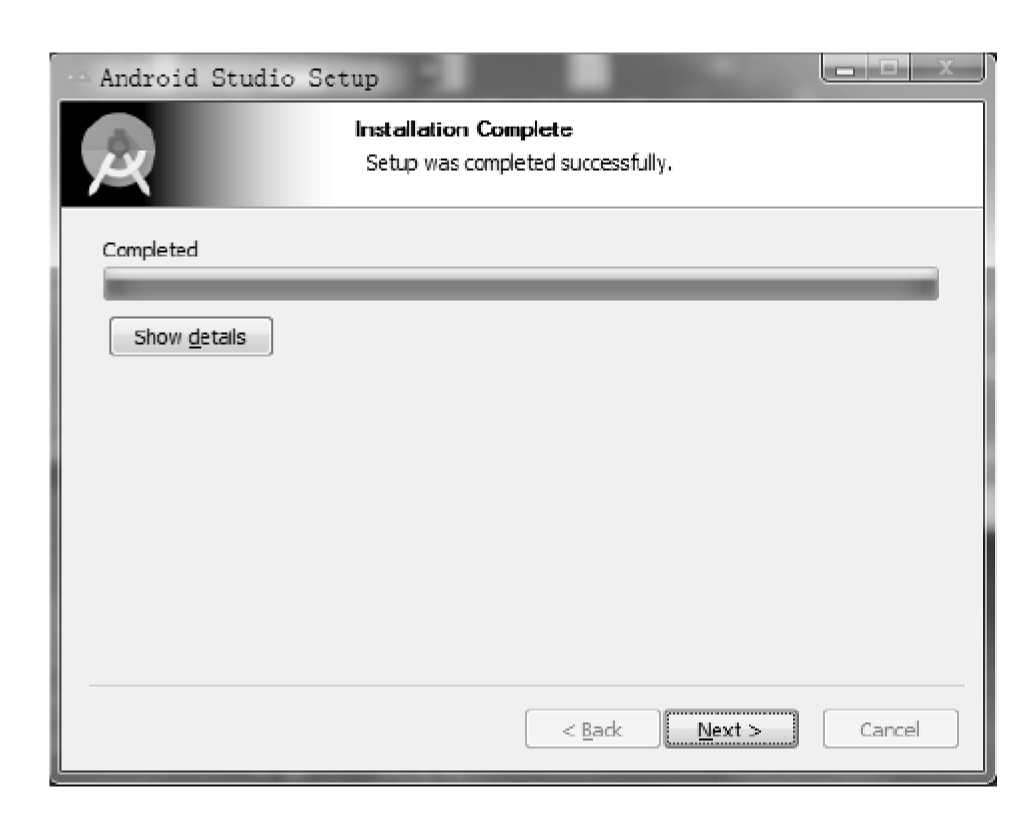


图 1-6　Android Studio 的安装结果对话框

安装完毕后弹出一个提示对话框，如图 1-7 所示，上面默认勾选了“Start Android Studio”，单击右下角的“Finish”按钮即可启动 Android Studio。



图 1-7　Android Studio 的安装完成对话框

1.2.2 启动 Android Studio

首次安装 Android Studio 3.0 会弹出一个提示对话框，如图 1-8 所示，在此开发者可以选择从哪个目录导入之前的 Android Studio 设置。为了更好地演示完整的启动过程，这里选择最下面的选项“Do not import settings”，表示不导入任何已有设置，完全重新开始进行设置。

选好之后，单击提示对话框下方的“OK”按钮，Android Studio 便执行启动操作，如图 1-9 所示。



图 1-8 Android Studio 首次安装后的设置导入提示对话框

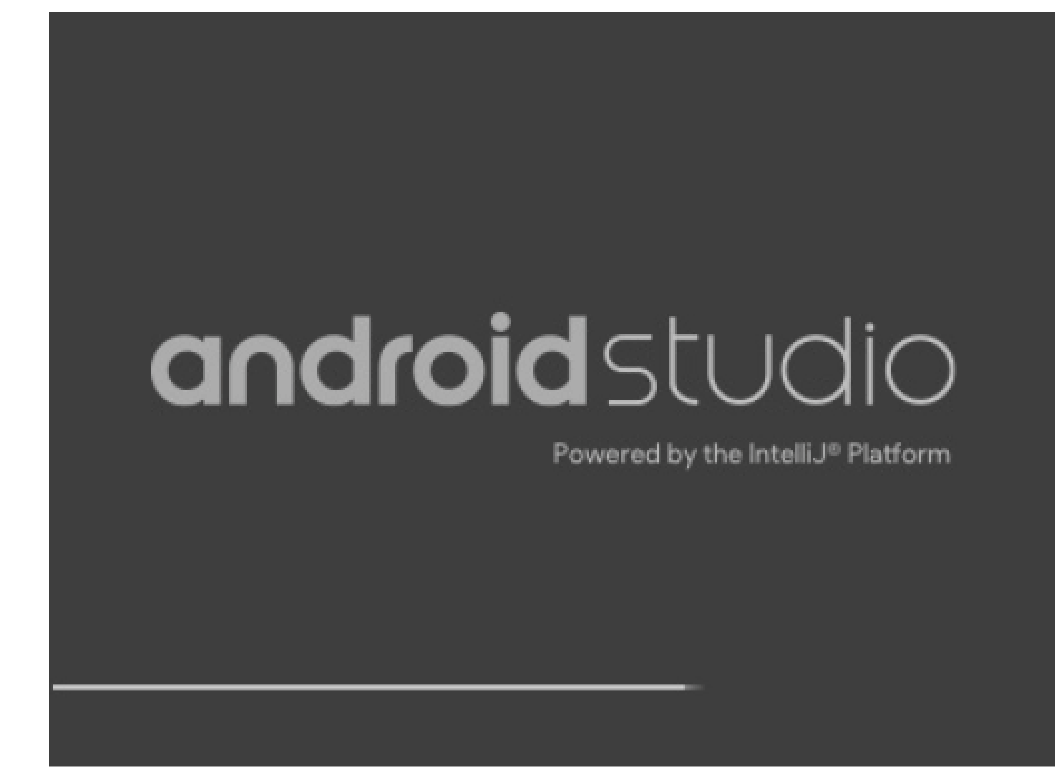


图 1-9 Android Studio 正在启动的提示对话框

因为前面选定了不导入任何设置重新开始，所以 Android Studio 将不会导入任何工程，而是弹出一个向导对话框，提示开发者去进行新的设置，如图 1-10 所示。单击“Next”按钮，进入下一页的安装类型对话框，如图 1-11 所示。

这里保持“Standard”选项，单击“Next”按钮，进入下一个对话框，如图 1-12 所示。继续单击“Next”按钮，进入向导的确认对话框，如图 1-13 所示。在该对话框确认 SDK 的安装路径是否正确，确认完毕单击“Finish”按钮，等待后续的 SDK 下载操作。

接下来的下载对话框会自动到谷歌网站更新组件，如图 1-14 所示。如果电脑本地没有 SDK，就继续等待下载更新，如果电脑本地已有现成的 SDK，就直接单击“Cancel”按钮取消下载，然后单击“Finish”按钮结束设置。最后弹出一个“Welcome to Android Studio”欢迎对话框，如图 1-15 所示。单击第一项的“Start a new Android Studio project”即可开始你的 Android 开发之旅。



图 1-10 Android Studio 的启动向导对话框 1

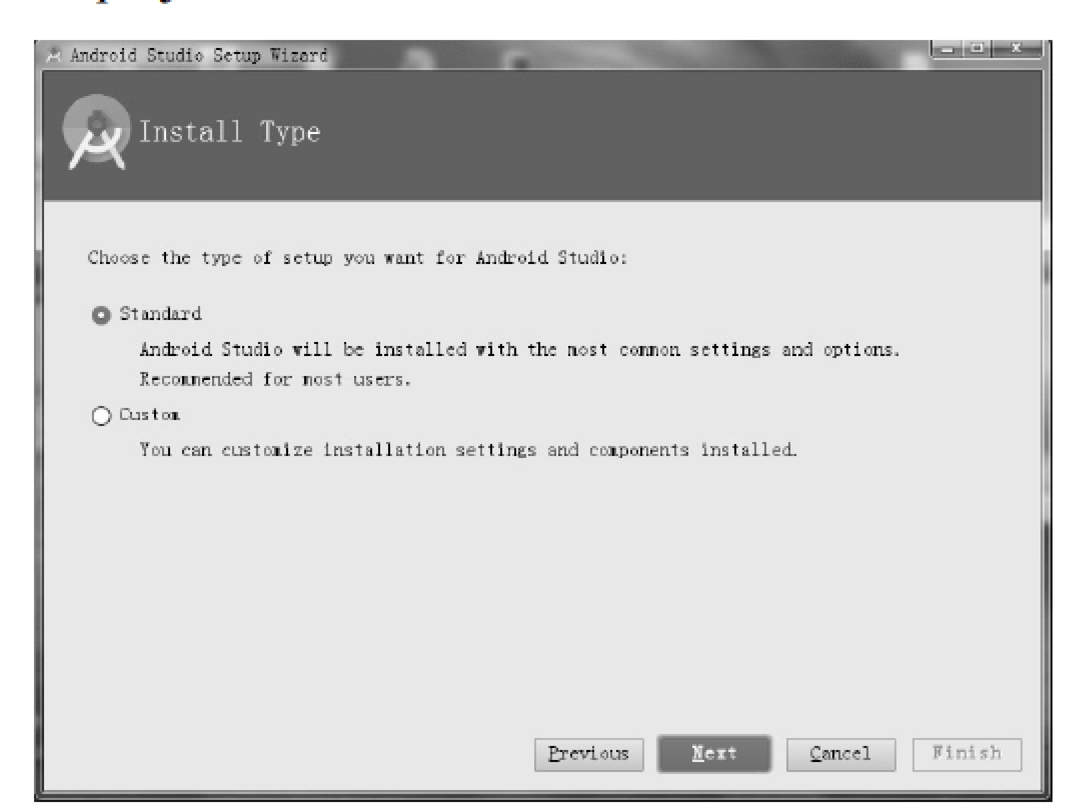


图 1-11 Android Studio 的启动向导对话框 2

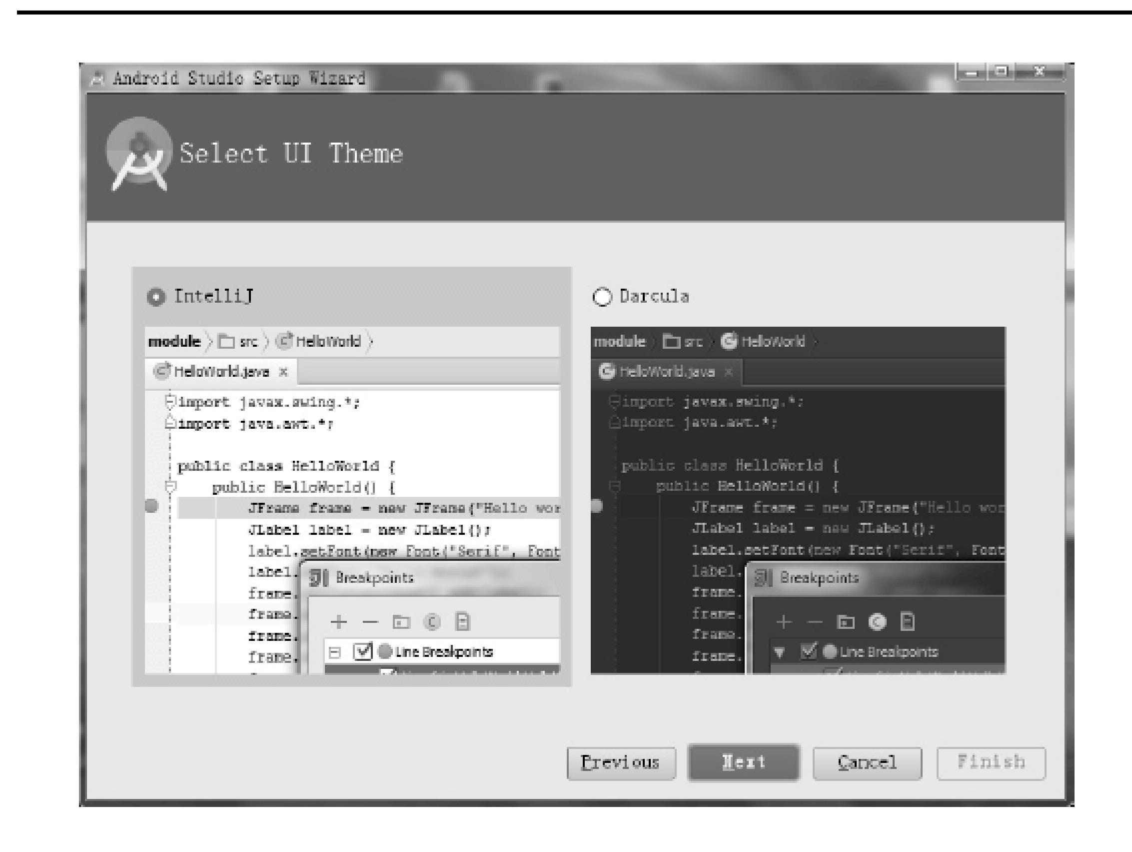


图 1-12　Android Studio 的启动向导对话框 3

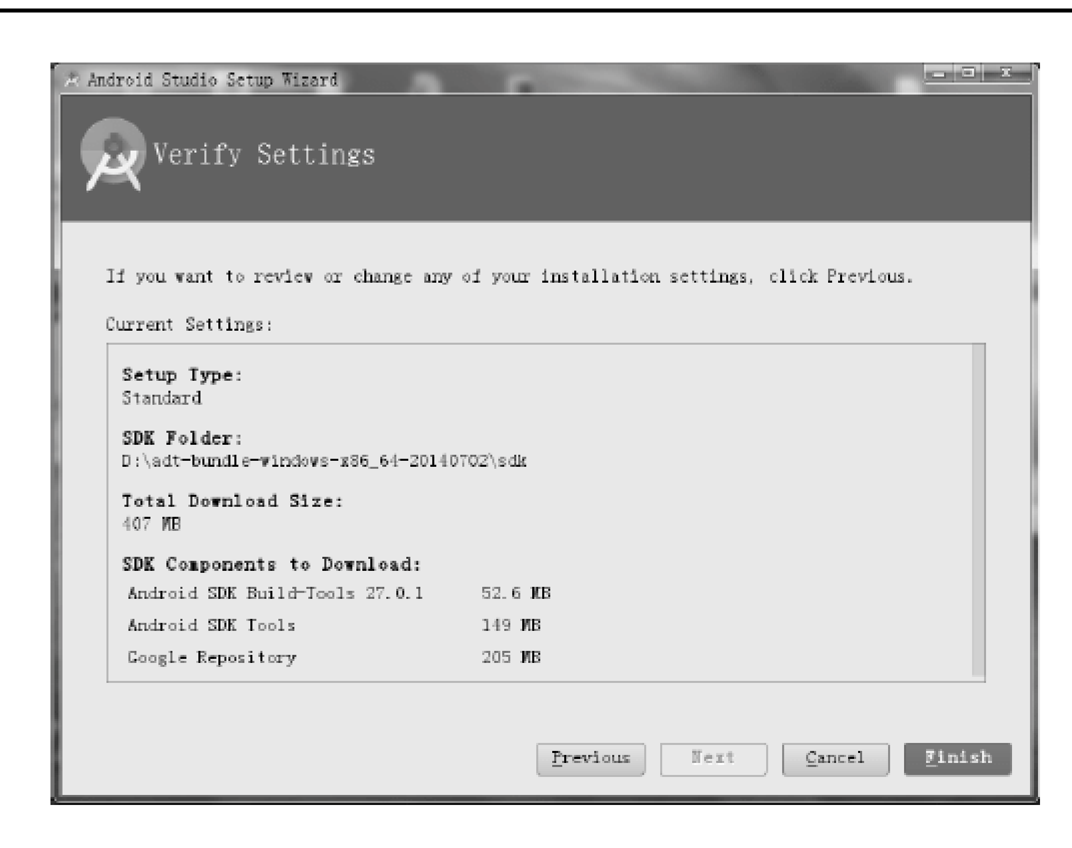


图 1-13　Android Studio 的启动向导对话框 4

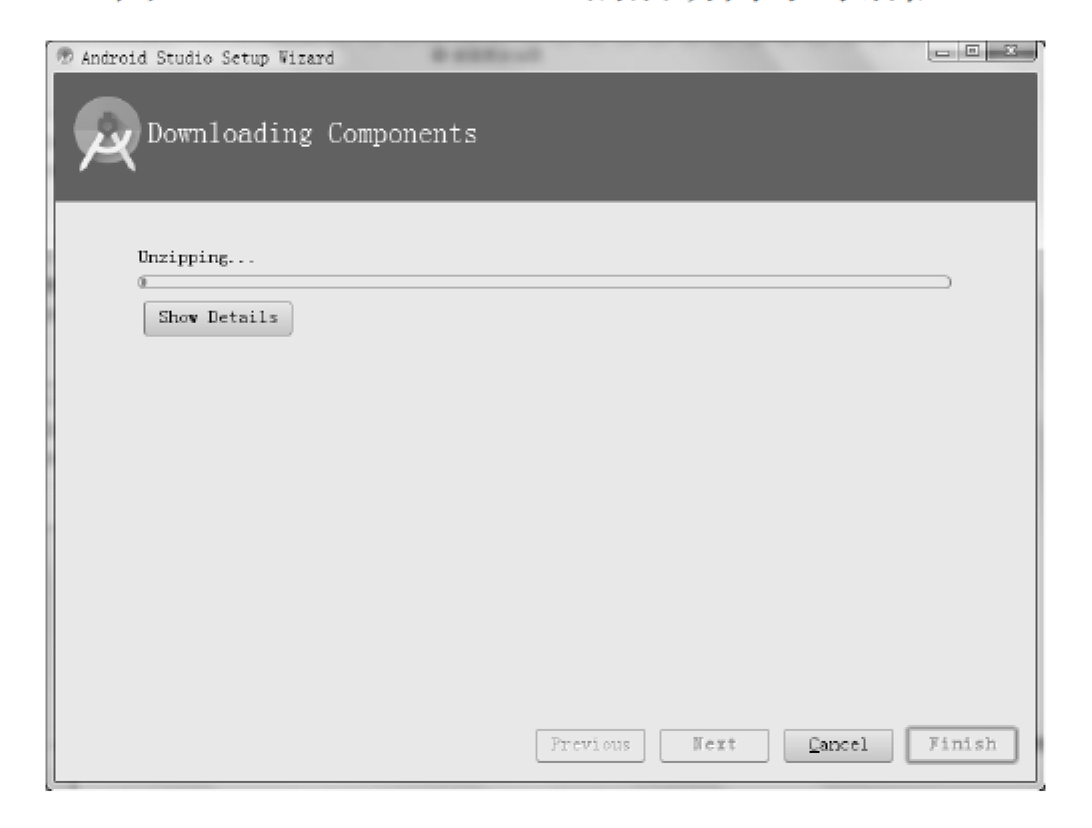


图 1-14　Android Studio 的启动向导对话框 5

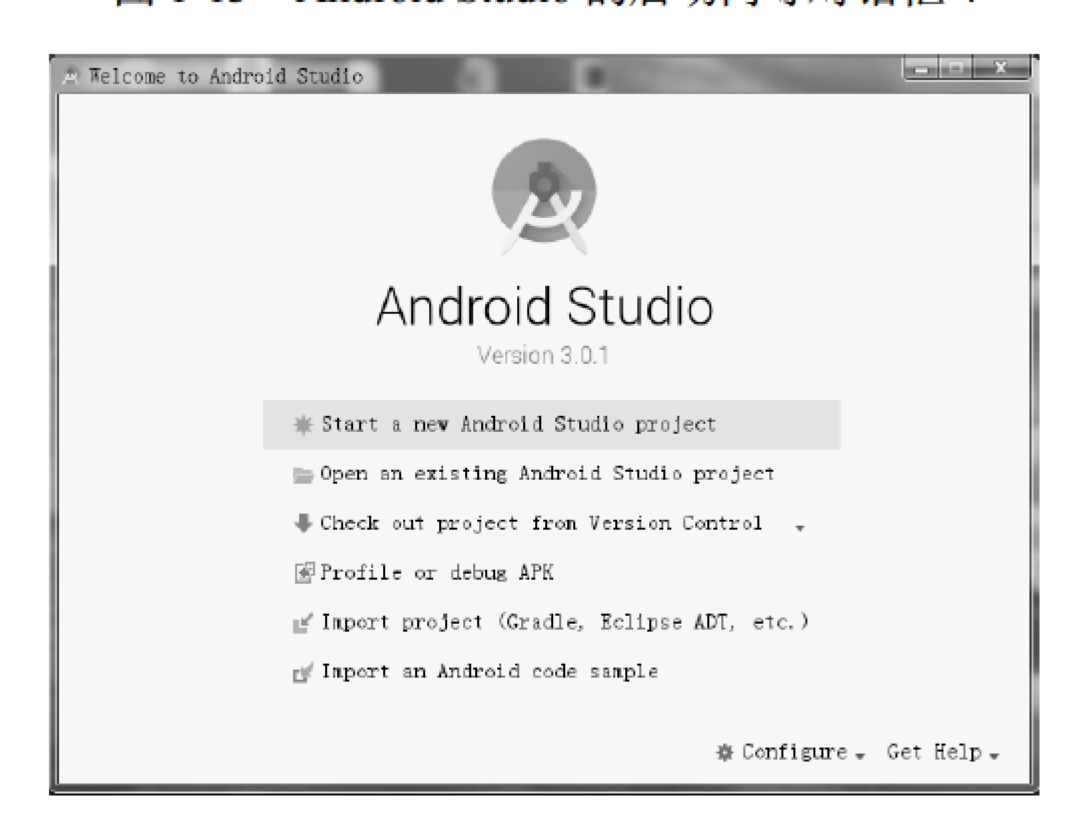


图 1-15　Android Studio 的启动欢迎对话框

1.2.3　创建 Kotlin 工程

1.2.2 小节提到 Android Studio 启动设置完成之后，会弹出欢迎对话框提示创建新的 Android 工程，此时单击第一项的“Start a new Android Studio project”打开工程创建对话框，如图 1-16 所示。

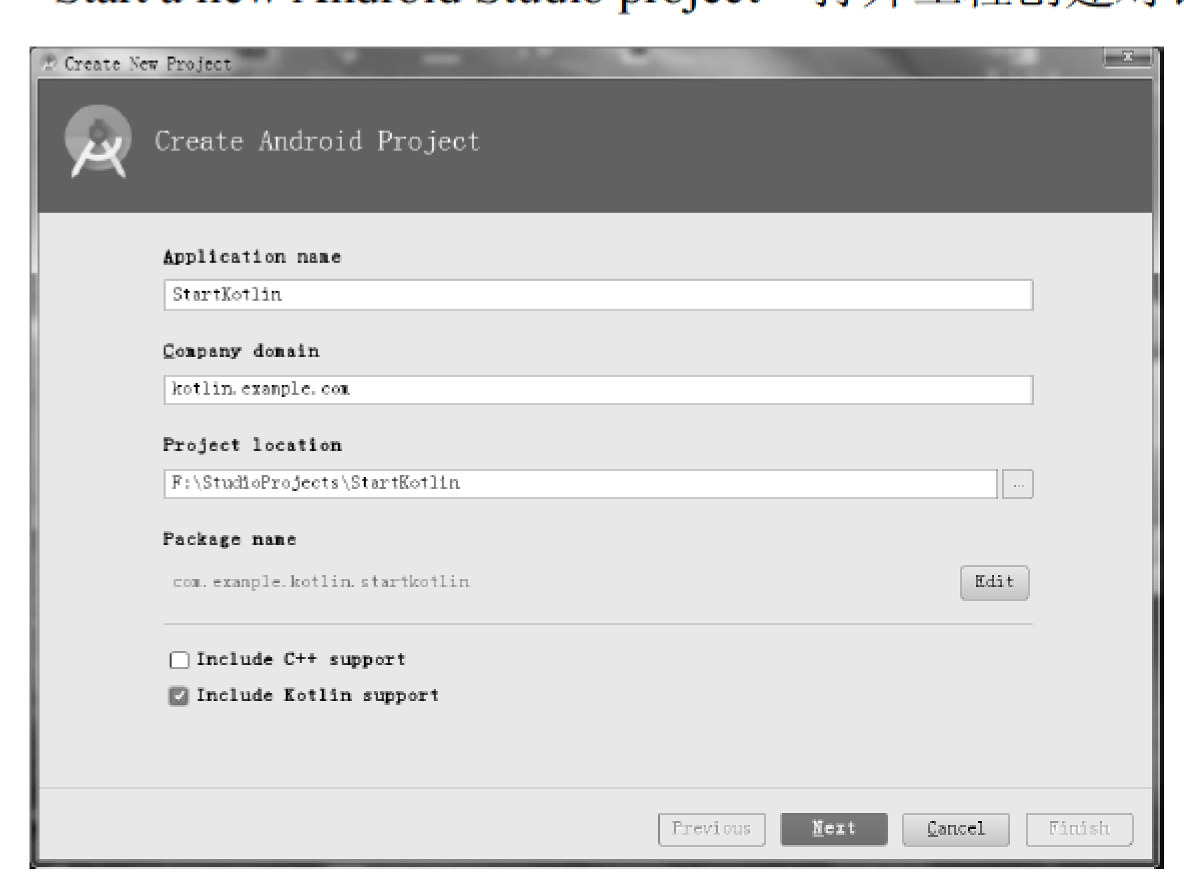


图 1-16　Android Studio 的工程创建对话框

在工程创建对话框中填写应用名称“Application name”以及公司域名“Company domain”，并选择或填写 Android 工程的本地保存路径“Project location”。注意，创建页面下方有两个选项“Include C++ support”和“Include Kotlin support”，其中勾选“Include C++ support”表示要进行 NDK/JNI 开发，但这不是本书的讲解范围，因此不必勾选该复选框；勾选“Include Kotlin support”则表示要进行 Kotlin 开发，因此务必勾选该复选框，才能继续后面的 Kotlin 开发学习。确认对话框中的各项信息都填写完毕，单击下方的“Next”按钮，进入目标设备对话框，如图 1-17 所示。

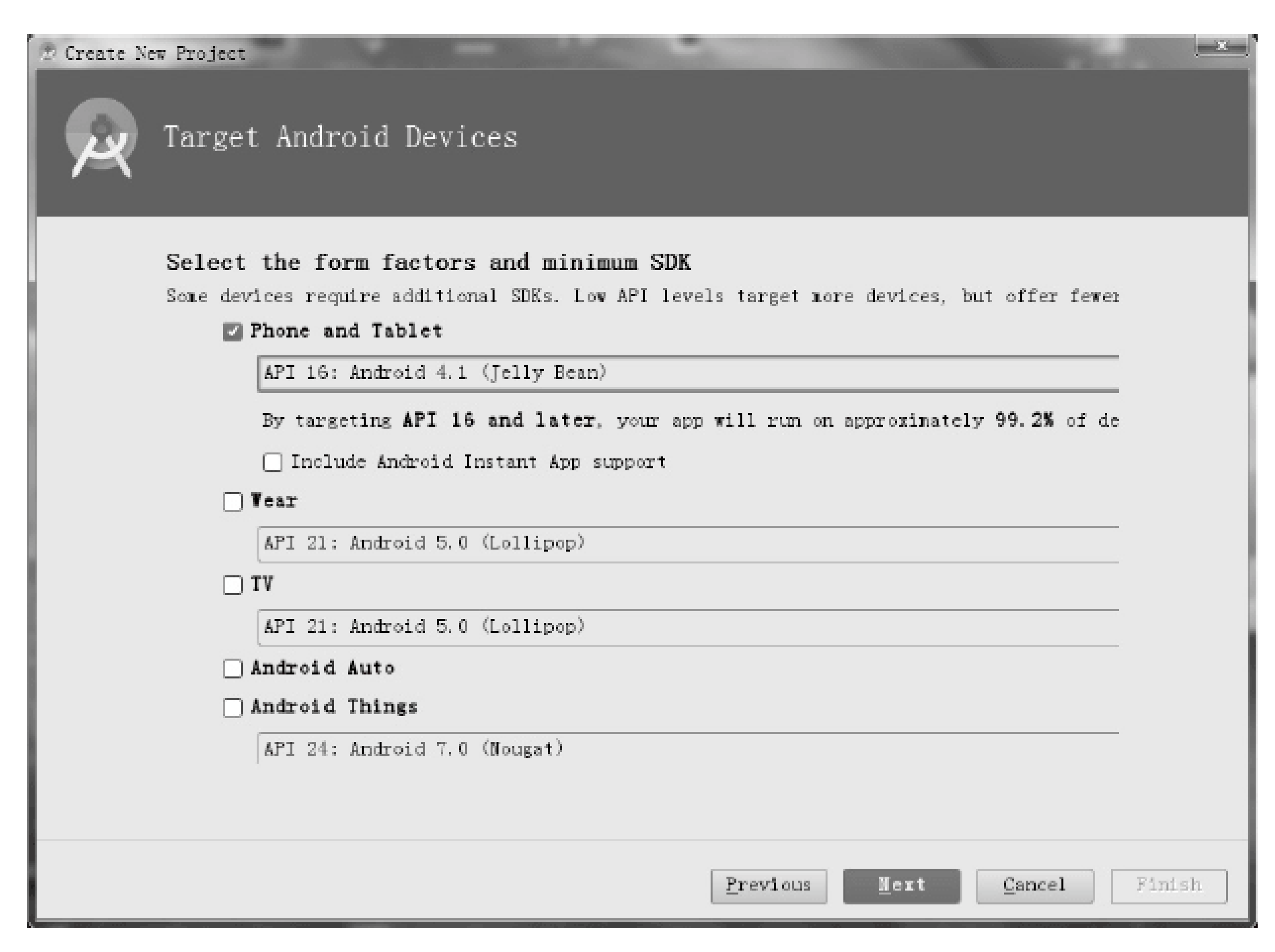


图 1-17　Android Studio 创建新工程时的目标设备对话框

在目标设备对话框中，Android Studio 默认勾选了“Phone and Tablet”，表示进行手机/平板应用开发，下面的 API 最低支持版本原本默认是 API 15，不过因为如通知的新特性从 API 16 开始才支持，所以这里建议把最低版本改为 API 16，接着单击“Next”按钮，进入初始风格对话框，如图 1-18 所示。



图 1-18　Android Studio 创建新工程时的初始风格对话框

在初始风格对话框中选择“Empty Activity”，然后单击下方的“Next”按钮，进入名称配置对话框，如图 1-19 所示。

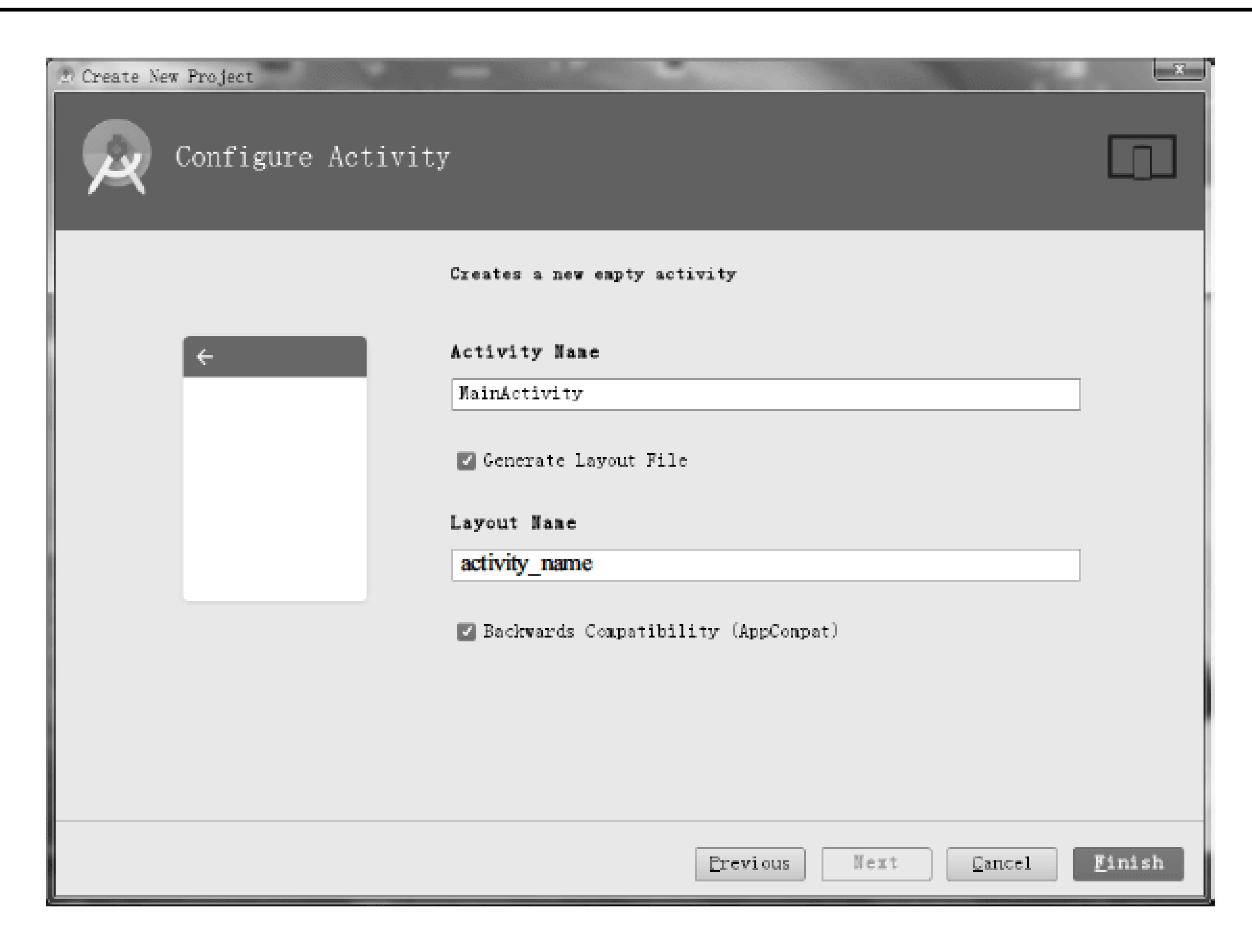


图 1-19　Android Studio 创建新工程时的名称配置对话框

在名称配置对话框保持默认设置，即活动代码名称“Activity Name”仍然填写“MainActivity”，布局文件名称“Layout Name”仍然填写“activity_name”，然后单击“Finish”按钮，进入 Android Studio 的完整开发界面，如图 1-20 所示。

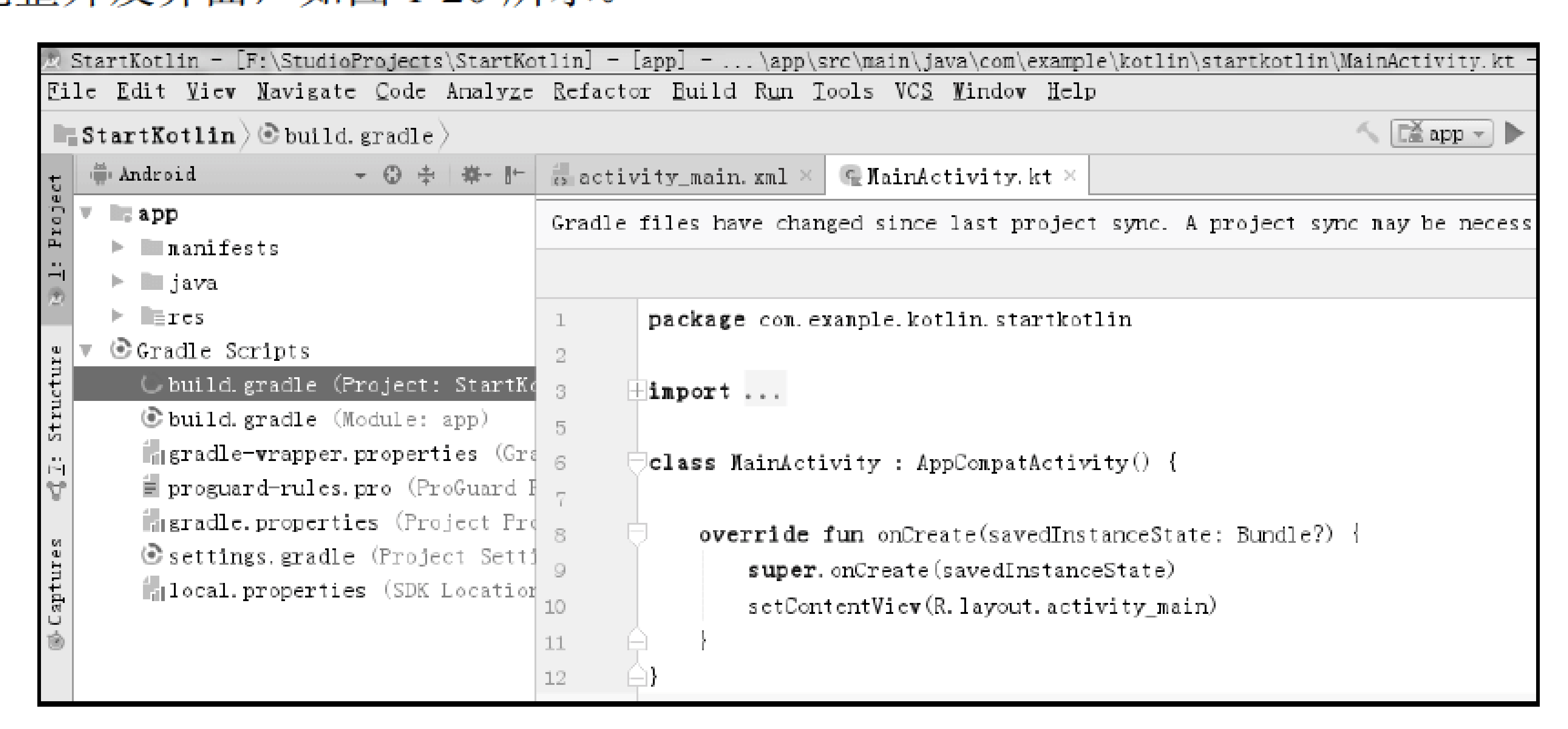


图 1-20　Android Studio 创建新工程之后的开发界面

在编写代码的时候，Android Studio 会自动编译。若开发者想手动重新编译，则有以下三种编译方式：

（1）选择菜单“Build”→“Make Project”，这个是编译整个项目下的所有模块。

（2）选择菜单“Build”→“Make Module ***”，这个是编译指定名称的模块。

（3）选择菜单“Build”→“Clean Project”，然后再选择菜单“Build”→“Rebuild Project”，这个是先清理项目，再对整个项目重新编译。

前面新创建的工程当然不会出现编译错误，直接运行就好了。先把手机通过数据线接入开发电脑的 USB 上，再依次选择菜单“Run”→“Run 'app'”，弹出目标设备选择对话框，列表中便会显示接入的手机设备，如图 1-21 所示。

单击“OK”按钮，执行测试 App 的安装操作，不出意外的话，一会儿即可在手机上看到测试应用的启动界面，具体效果如图 1-22 所示。

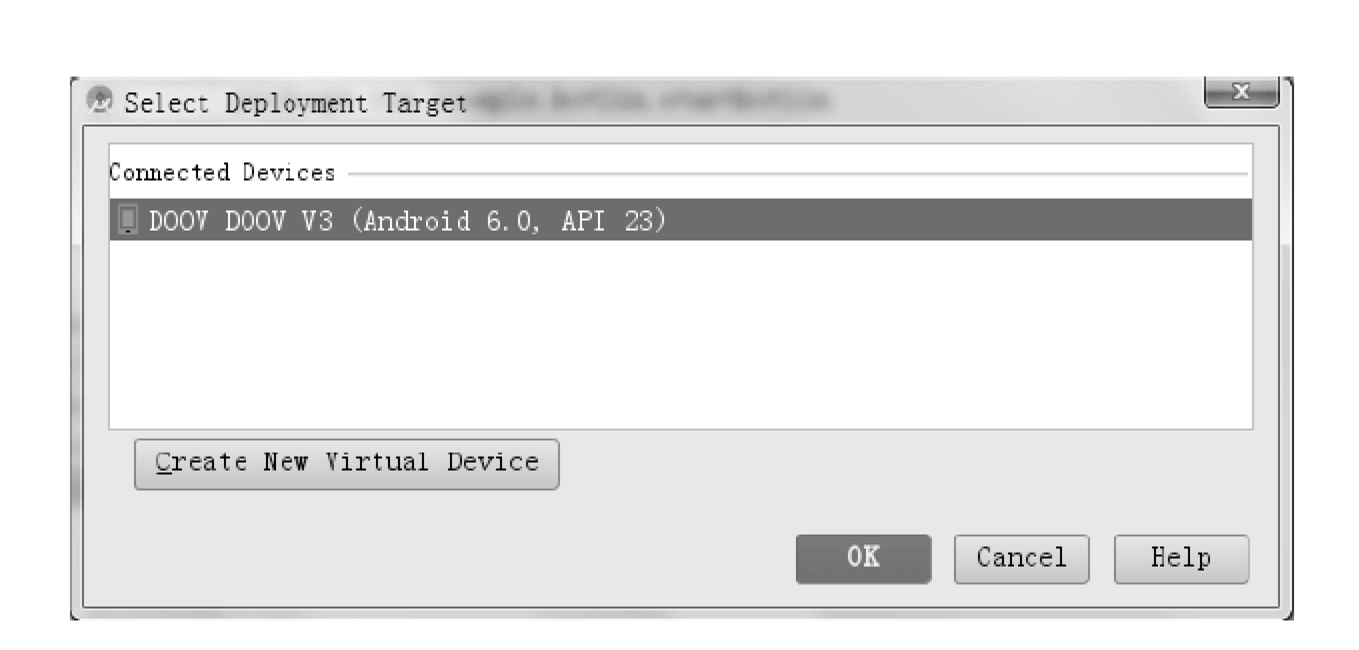




图 1-21　运行测试应用时弹出的目标设备选择对话框

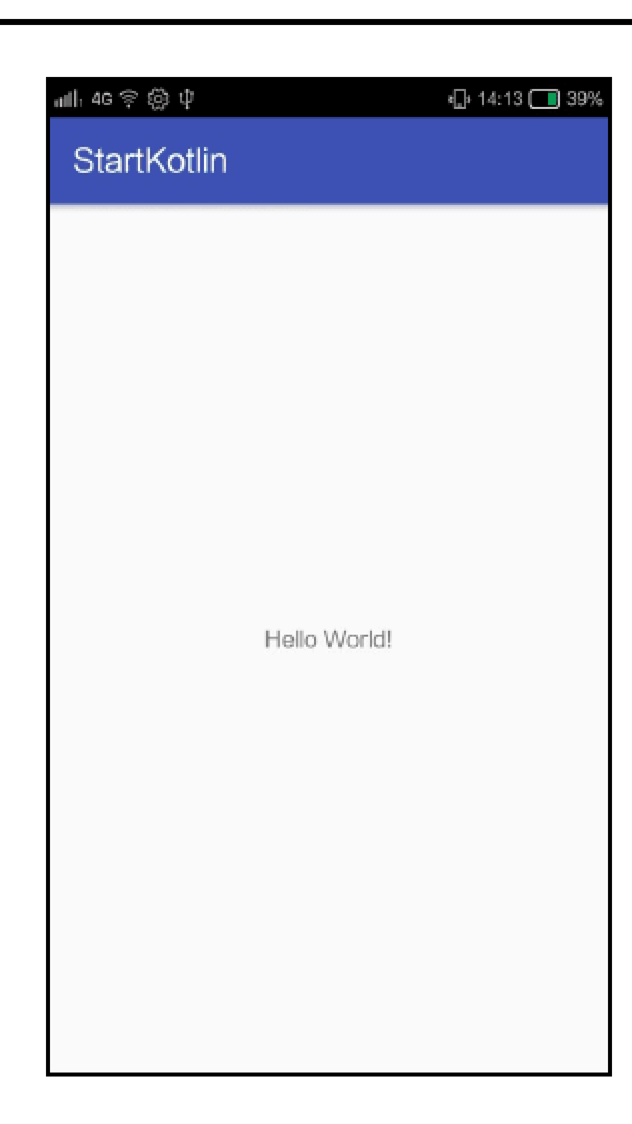


图 1-22　测试应用的启动界面

1.2.4　新建 Kotlin 文件

上一小节创建 Kotlin 工程后主要生成两个文件，一个是 Kotlin 代码文件 MainActivity.kt，另一个是 XML 布局文件 activity_name.xml。其中，MainActivity.kt 就是扩展名为 kt 的 Kotlin 格式的代码文件，相对应地，Java 代码文件的扩展名为 java。这个 MainActivity.kt 是在创建工程时自动生成的，那么如何在已有工程中创建新的 Kotlin 文件呢？Kotlin 代码文件可以分为两类：普通的 Kotlin 文件、用于页面 Activity 的文件，这两类 Kotlin 文件拥有各自的创建方式，具体说明如下。

1. 普通的 Kotlin 文件创建

右击待创建文件的包名，在弹出的快捷菜单中依次选择“New”→“Kotlin File/Class”，菜单界面如图 1-23 所示。

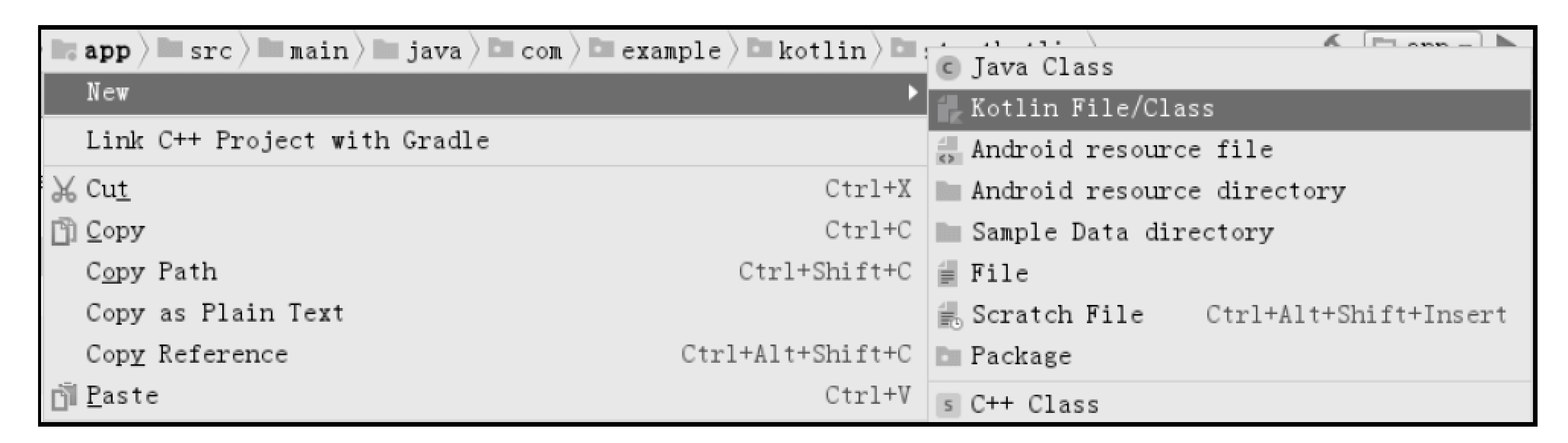


图 1-23　通过快捷菜单创建 Kotlin 文件

也可在顶部的主菜单栏上依次选择“New”→“Kotlin File/Class”，菜单界面如图 1-24 所示。

上述两种方式都会打开 Kotlin 文件的创建对话框，如图 1-25 所示。这里在“Name”输入框填写文件名称，在“Kind”下拉框中可单击弹出下拉列表，如图 1-26 所示，根据要求可选择对应的文件类型（File 表示普通文件，Class 表示类文件，Interface 表示接口文件，Enum class 表示枚举文件，Object 表示对象文件），然后单击“OK”按钮完成文件创建操作。

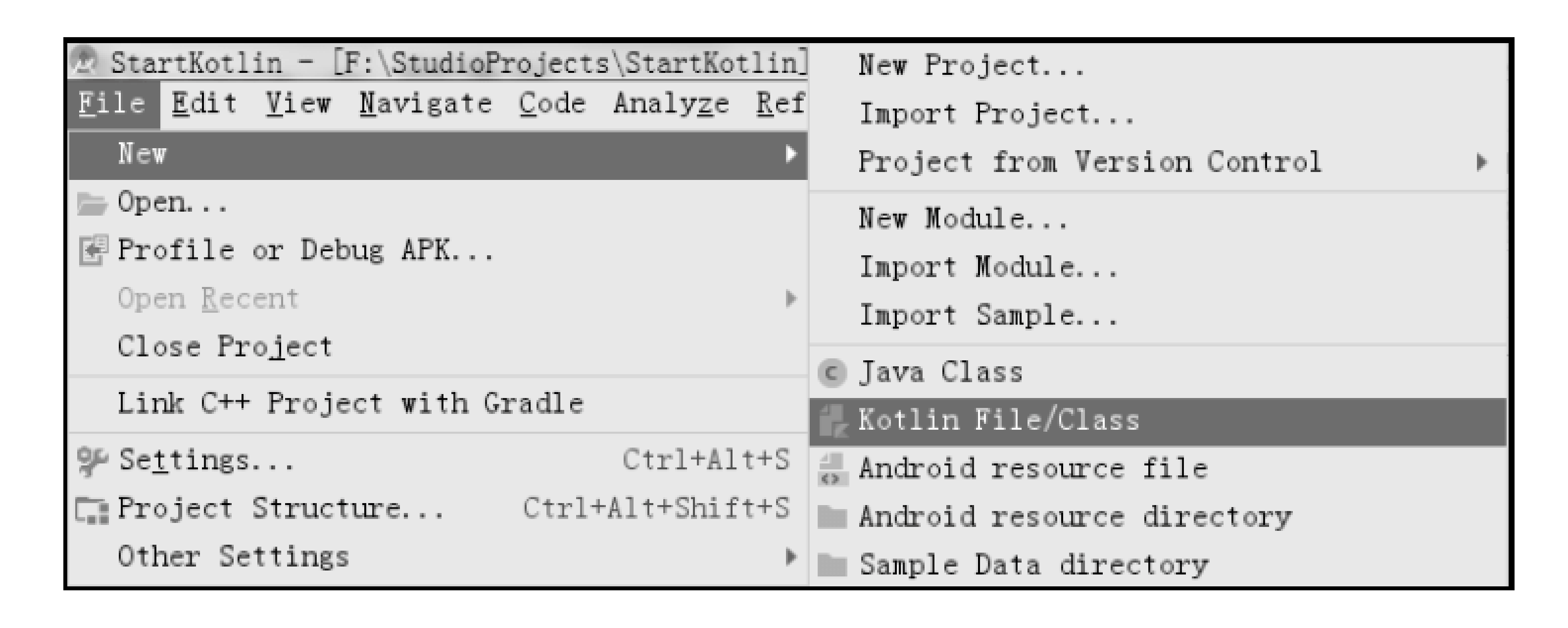


图 1-24　通过主菜单栏创建 Kotlin 文件

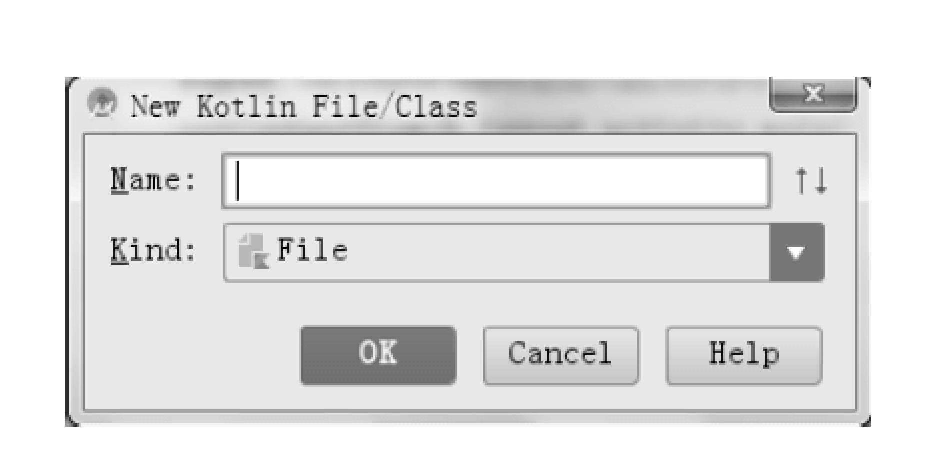


图 1-25　Kotlin 文件的创建对话框

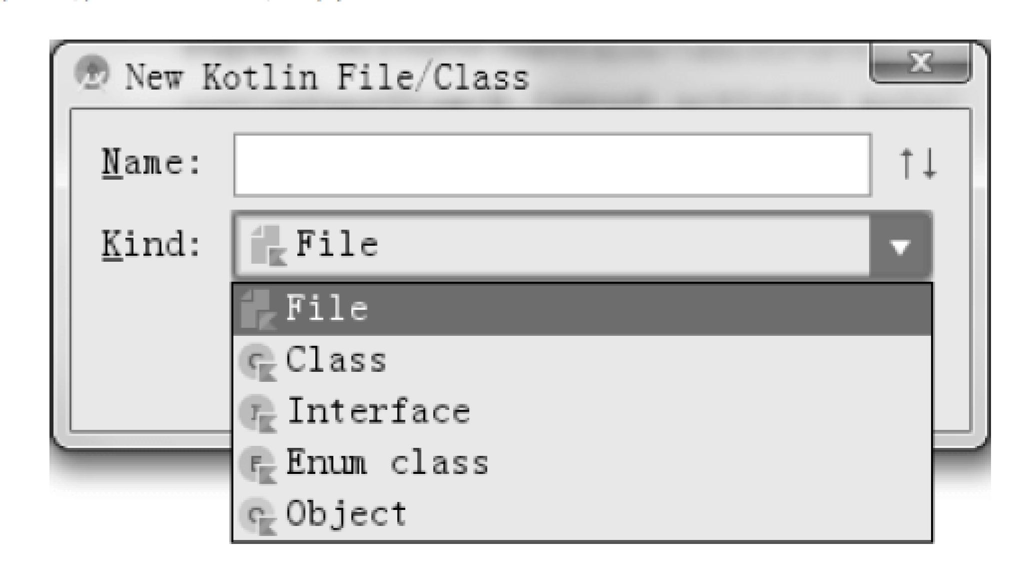


图 1-26　选择 Kotlin 文件的类型

2. 用于页面 Activity 的文件创建

选中待创建文件的包名，在顶部的主菜单栏上依次选择“New”→“Activity”→“Empty Activity”，菜单界面如图 1-27 所示。

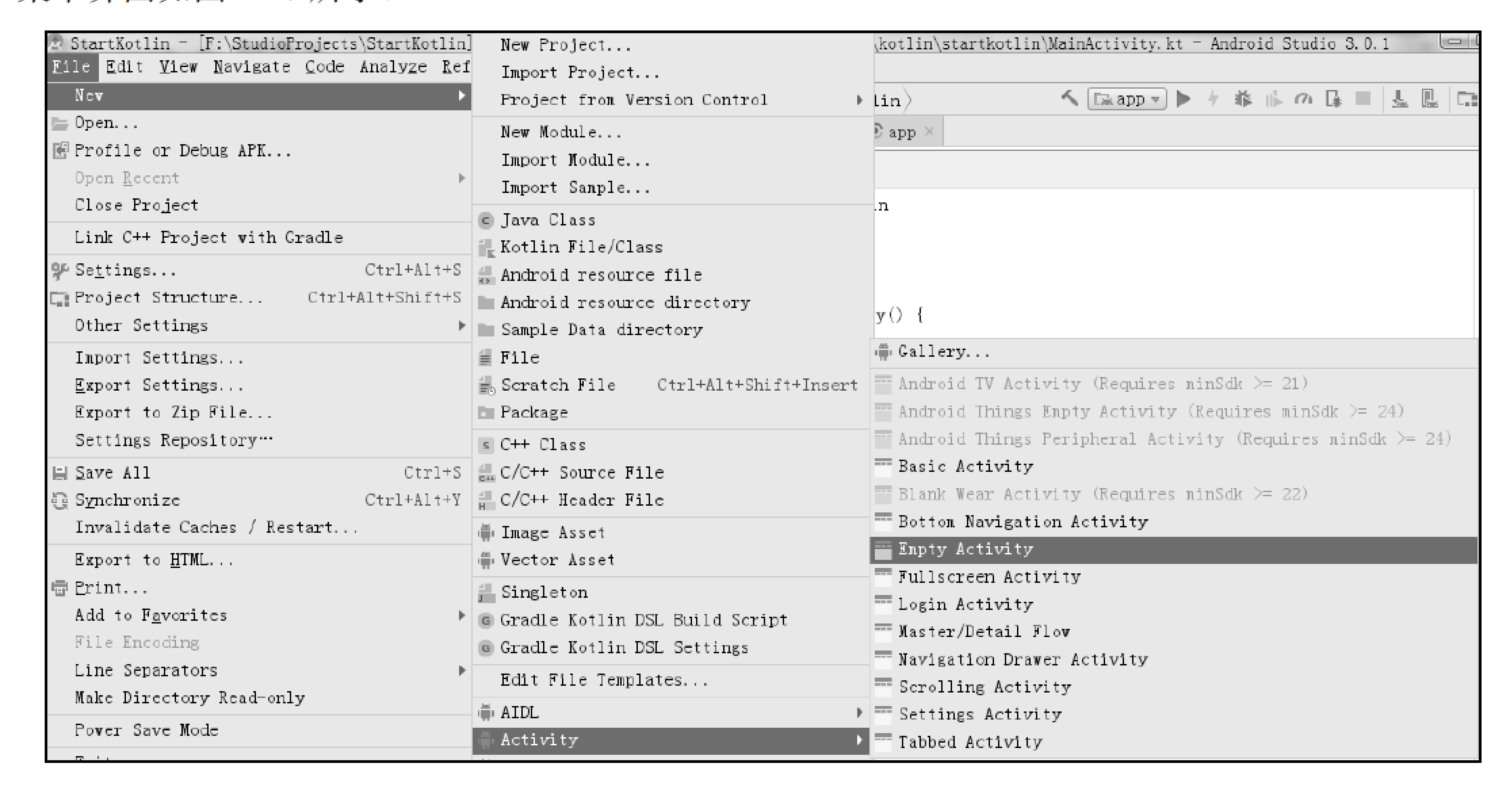


图 1-27　创建用于页面 Activity 的文件

然后打开活动 Activity 的创建对话框，如图 1-28 所示，分别在“Activity Name”和“Layout Name”两个输入框中填写活动名称与布局名称，并在下方的源码语言“Source Language”的下拉列表中选择“Kotlin”，表示新创建的 Activity 代码采用 Kotlin 编码。

确认好新 Activity 的创建信息，单击“Finish”按钮，Android Studio 便会自动创建 Kotlin 代码文件 MainActivity2.kt，以及与之对应的 XML 布局文件 activity_name2.xml。创建后的工程文件结构如图 1-29 所示。



图 1-28　Activity 页面文件的创建对话框

图 1-29　创建新 Activity 文件后的工程目录结构

其实，Kotlin 的文件创建还是很简单的，掌握这些基本的文件操作可以为后面的工具使用打好基础。

1.3　SDK 安装与插件升级

本节主要介绍 Android Studio 3.0 环境对 SDK 和插件的安装升级说明，包括如何安装最新的 SDK、如何升级 Gradle 插件、如何把 Kotlin 插件升级到最新版本等。

1.3.1　安装最新版 SDK

由于目前官方的 Android Studio 3.0 安装包没有自带 SDK，安装过程也只会去下载 Android 8.0 的 SDK（API 26），因此如果读者是第一次安装 Android Studio，就得自己另外安装最新版本的 SDK。首先打开 Android Studio，在界面右上角的工具栏中找到“SDK Manager”的图标，如图 1-30 所示。

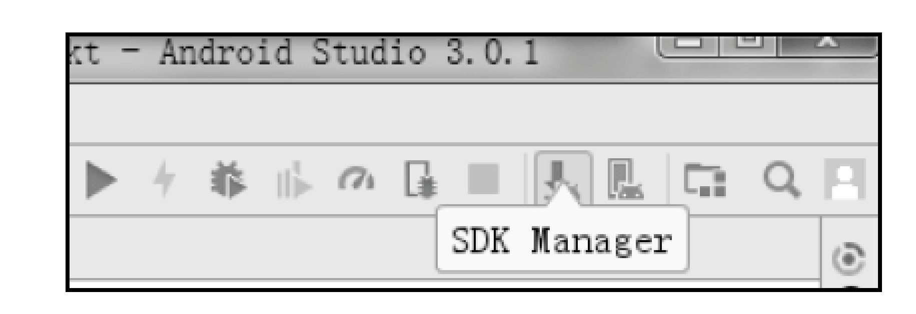


图 1-30　在工具栏中找到“SDK Manager”

单击该图标，打开 Default Settings 对话框，如图 1-31 所示，在对话框右侧的 SDK 列表中勾选最上面的 SDK 版本，比如“Android API 27”，然后单击右下角的“Apply”按钮，命令 Android Studio 执行该版本 SDK 的下载操作。

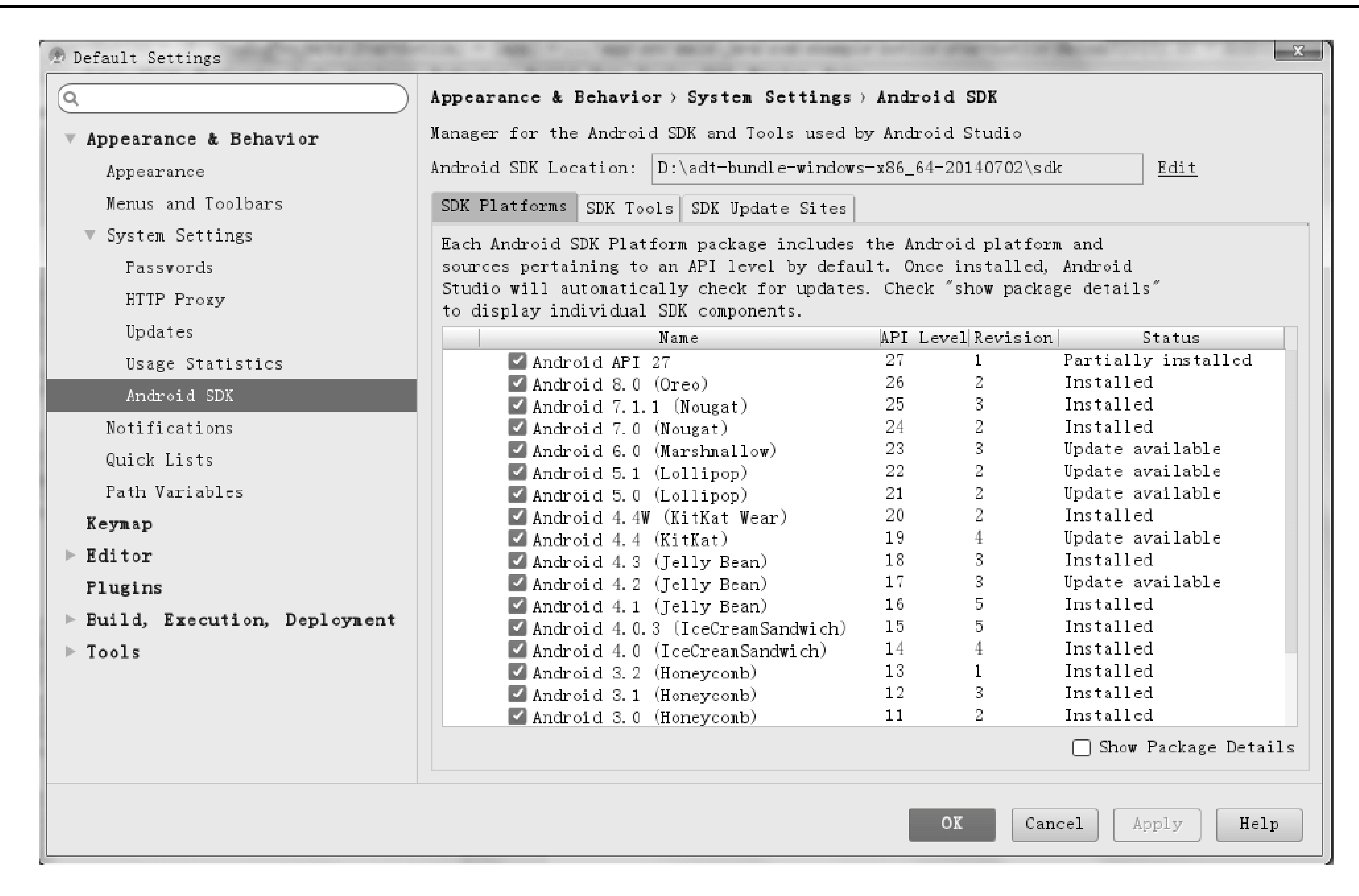


图 1-31　SDK 管理器的版本管理对话框

等待 Android Studio 下载并安装最新版本的 SDK 之后，重启 Android Studio，即可正常使用该版本的 SDK 编译工程。

1.3.2　升级 Gradle 插件

Android Studio 3.0 支持的 Gradle 插件版本至少为 4.1，然而通常 App 工程自带的插件版本不能满足要求，使得 Android Studio 3.0 打开已有工程时往往要重新下载最新的 Gradle 插件，造成漫长的等待时间。与其让 Android Studio 老牛破车般地下载 Gradle，不如自己动手将最新版的 Gradle 插件下载到本地，然后重新配置 Gradle 插件目录。具体步骤如下：

步骤 01 打开电脑上的下载软件，输入 Gradle 4.1 的下载地址“http://downloads.gradle.org/distributions/gradle-4.1-all.zip”，把这个 4.1 的压缩包下载到电脑本地，并解压该压缩包到指定目录，比如“D:\Android\gradle-4.1”。

步骤 02 打开 Android Studio，依次选择菜单“File”→“Settings”，打开设置对话框，在对话框左侧的菜单列表再依次选择“Build, Execution, Deployment”→“Gradle”，此时对话框右侧展示 Gradle 插件的配置界面，如图 1-32 所示。

步骤 03 在 Gradle 配置界面上选中“Use local gradle distribution”，并在下方的“Gradle home:”输入框中填写前面 gradle-4.1-all.zip 的解压路径，例如“D:\Android\gradle-4.1”。

步骤 04 单击 Gradle 配置界面右下方的“OK”按钮，完成 Gradle 插件的路径配置。

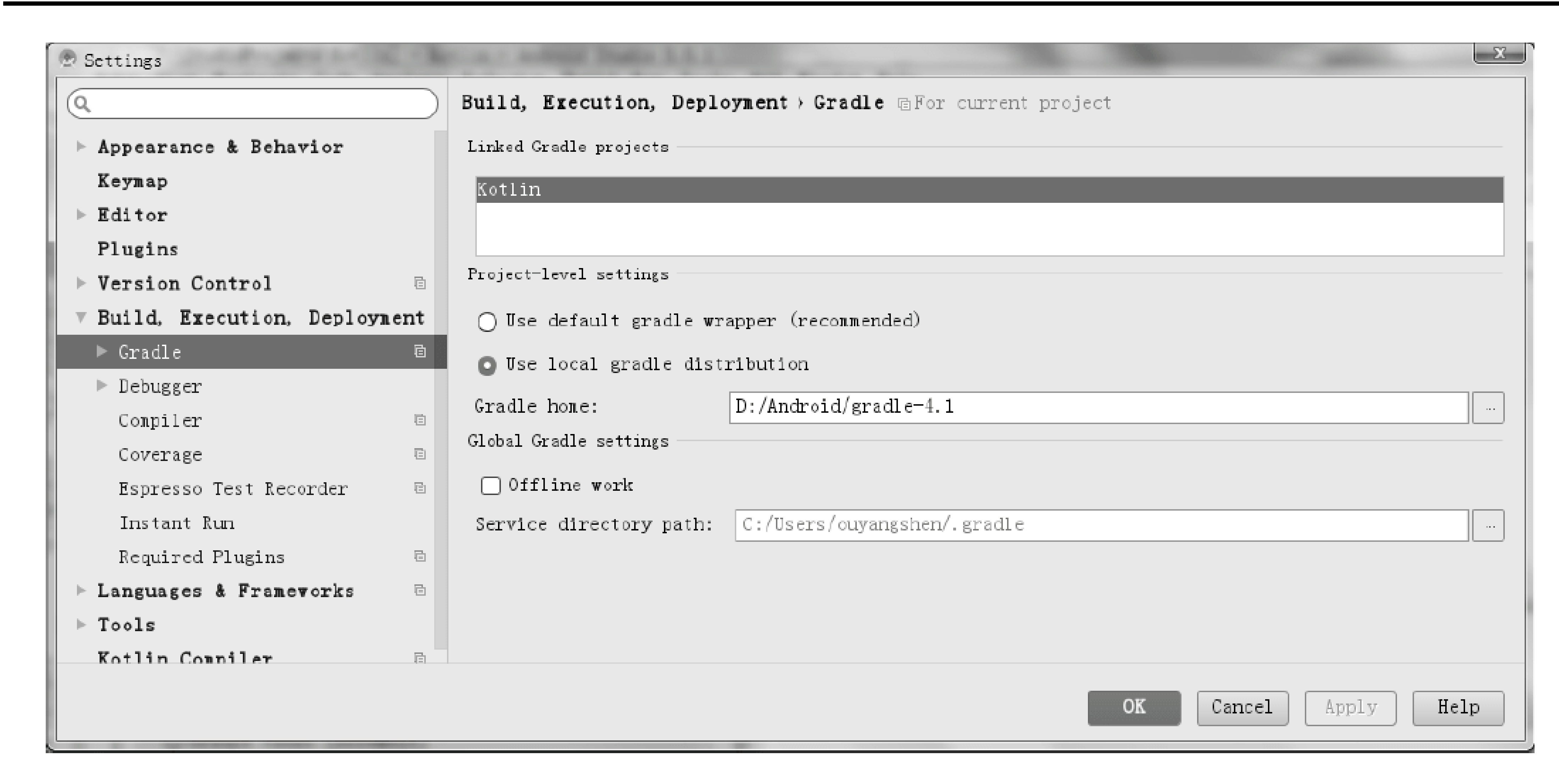


图 1-32 Gradle 插件的配置界面

1.3.3 升级 Kotlin 插件

Android Studio 虽然从 3.0 开始集成了 Kotlin 开发环境，但只是内置了某个版本的 Kotlin 插件。比如 Android Studio 3.0.1 集成的 Kotlin 插件版本为 1.1.51，随着 Kotlin 语言的更新换代，它的插件也得跟着升级。如何在 Android Studio 上手动升级 Kotlin 插件呢？且看下面的具体步骤说明。

步骤 01 依次选择菜单“File”→“Settings”，在弹出窗口右边的输入框中填写“kotlin”，从已安装插件里筛选出 Kotlin 插件，如图 1-33 所示，可见此时默认安装的 Kotlin 插件版本为 1.1.51。

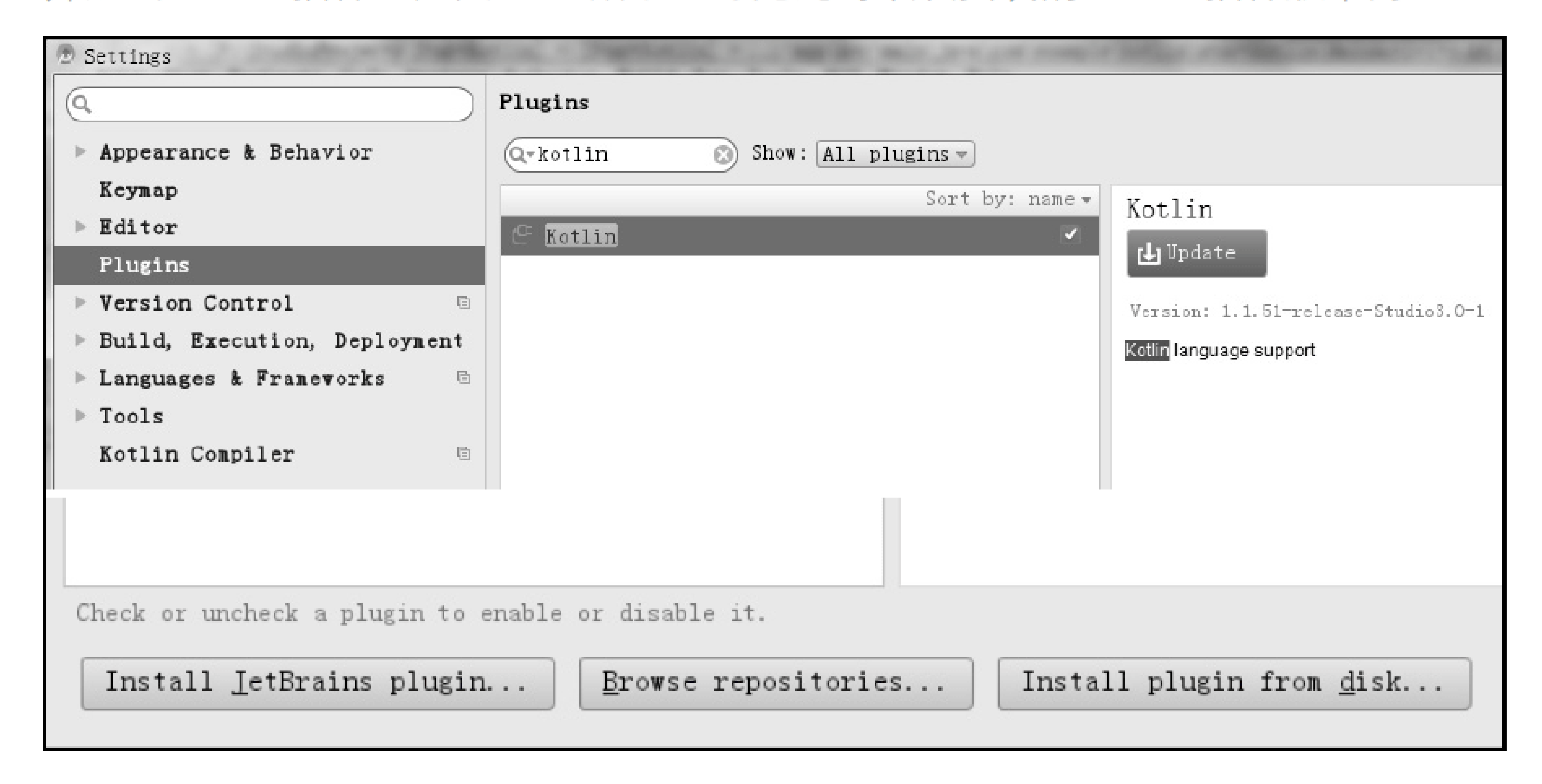


图 1-33 在已安装插件库中找到 Kotlin 插件

步骤 02 插件设置下方有一排三个按钮，单击左边的“Install JetBrains plugin...”按钮，打开远程的插件资源窗口。在该窗口左上角的输入框中填写“Kotlin”，筛选出符合条件的插件列表，如图 1-34 所示。

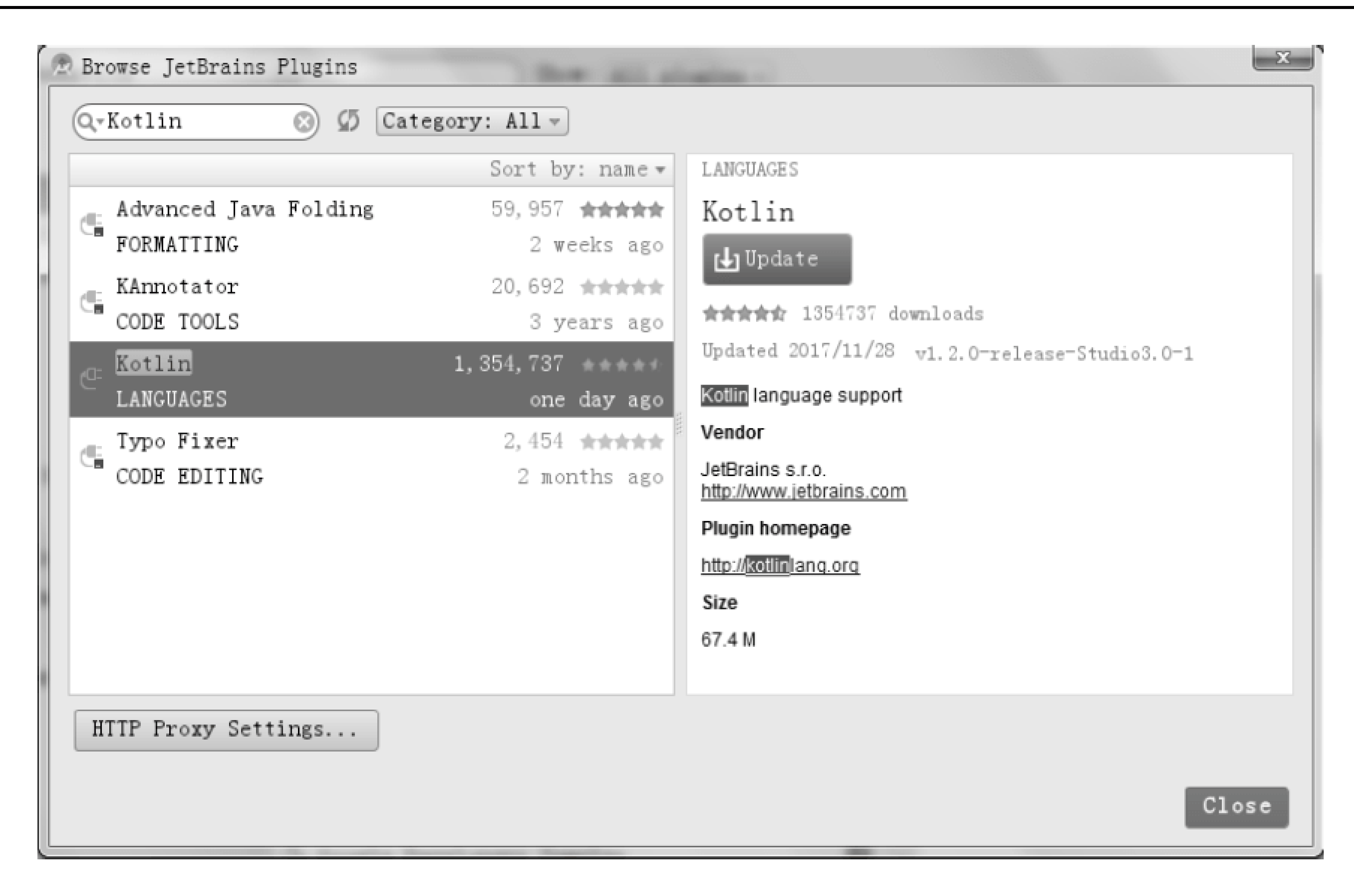


图 1-34　在远程资源库中找到最新的 Kotlin 插件

步骤 03 可见下方的插件列表会定位到符合搜索条件的插件位置，单击“Kotlin”（LANGUAGES），窗口右侧就会展示 Kotlin 插件的详细信息。发现远程插件库中的 Kotlin 最新版本为 2017 年 11 月 28 日推出的 1.2 版本，单击窗口右边的“Update”按钮执行升级操作。接着 Android Studio 开始自动下载 Kotlin 插件，下载过程如图 1-35 所示。

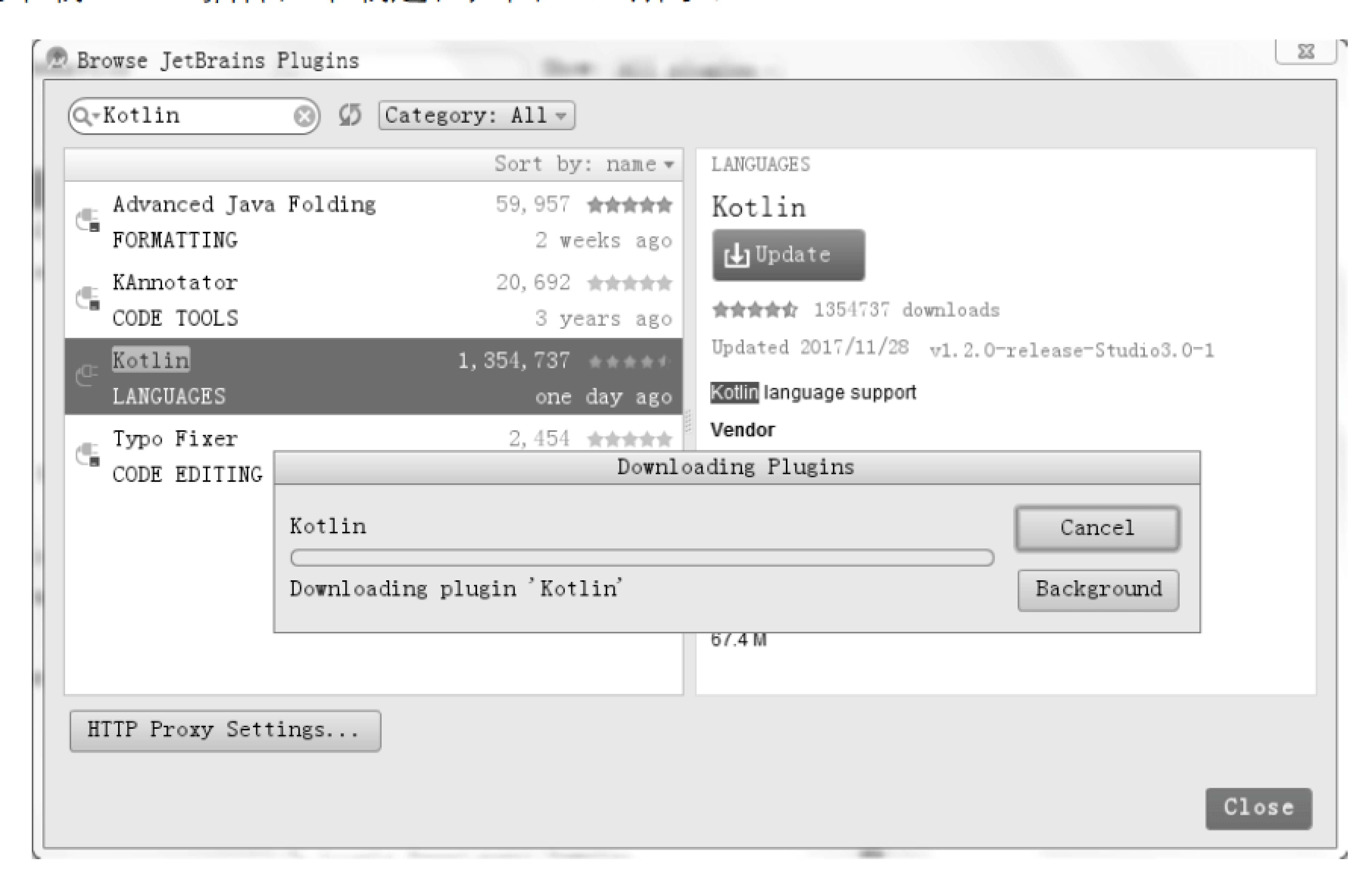


图 1-35　升级 Kotlin 插件的下载弹窗

步骤 04 等待 Kotlin 下载并更新完毕，此时原来的“Update”按钮变成了“Restart Android Studio”按钮，提示需要重启 Android Studio 使新插件生效，如图 1-36 所示。

步骤 05 根据提示关闭 Android Studio，再次启动 Android Studio，即可在 Android Studio 使用最新版本的 Kotlin 插件。

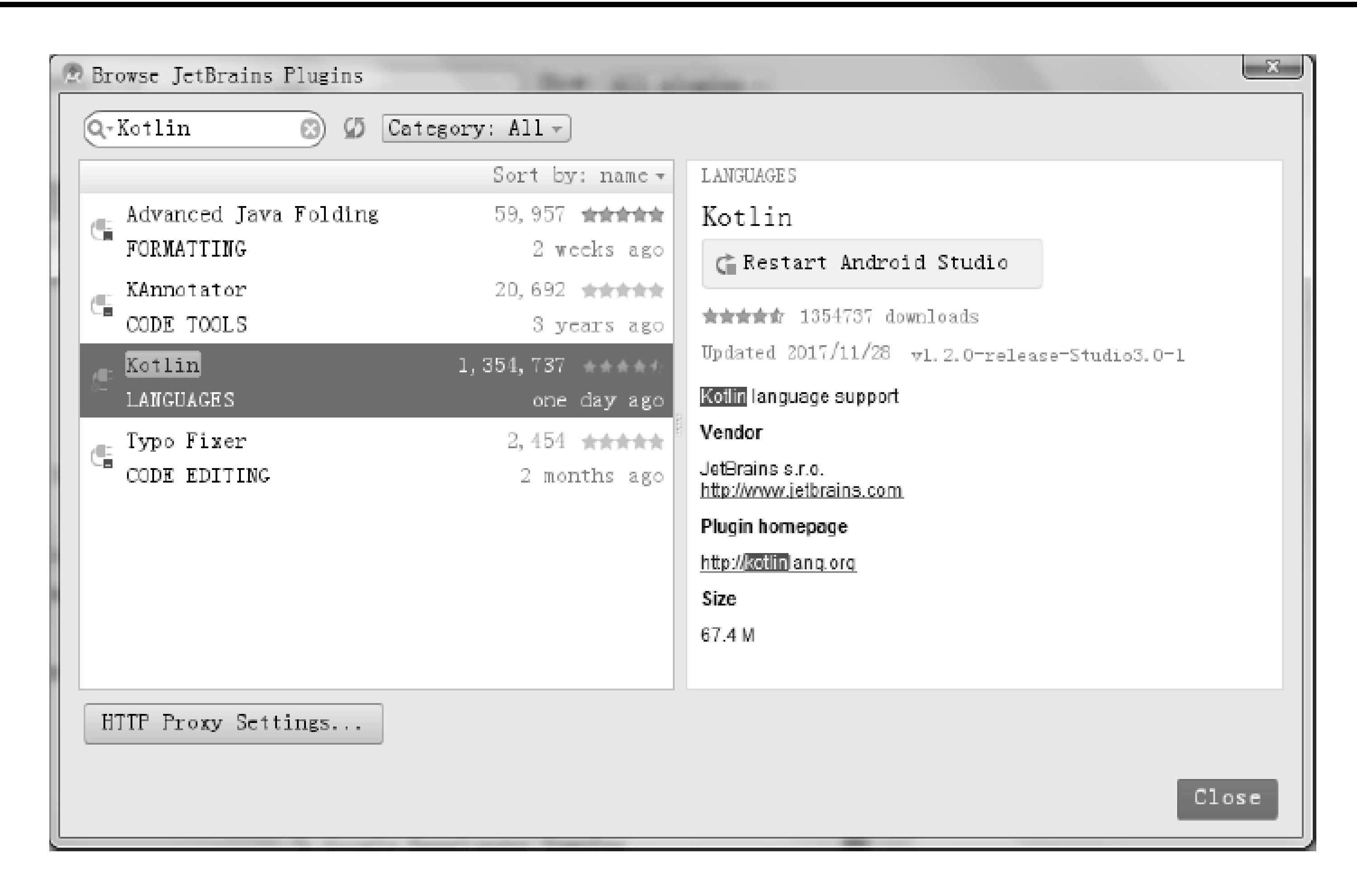


图 1-36　Kotlin 下载并更新完毕后的插件窗口

1.4　Kotlin 简单配置

本节主要介绍 Android Studio 3.0 环境对 Kotlin 的编译配置说明，包括如何通过菜单调整 Kotlin 编译配置、如何手工修改编译配置文件、如何将 Java 代码转换成 Kotlin 代码等。

1.4.1　调整 Kotlin 编译配置

1.3.3 小节介绍了如何将 Kotlin 插件升级到最新版本，不过 App 工程采取的 Kotlin 编译版本不一定跟最新版本一致。因为 Kotlin 允许指定使用某个低版本来编译工程，就像 Java 即使已经推出 1.9 版本，也能使用 1.8、1.7 甚至 1.6 来编译 Java 工程。

调整 App 工程的 Kotlin 编译版本很简单，依次选择菜单“File”→“Settings”，在打开的窗口左侧菜单列表选中“Kotlin Compiler”，窗口右边便会打开 Kotlin 编译配置界面，如图 1-37 所示。

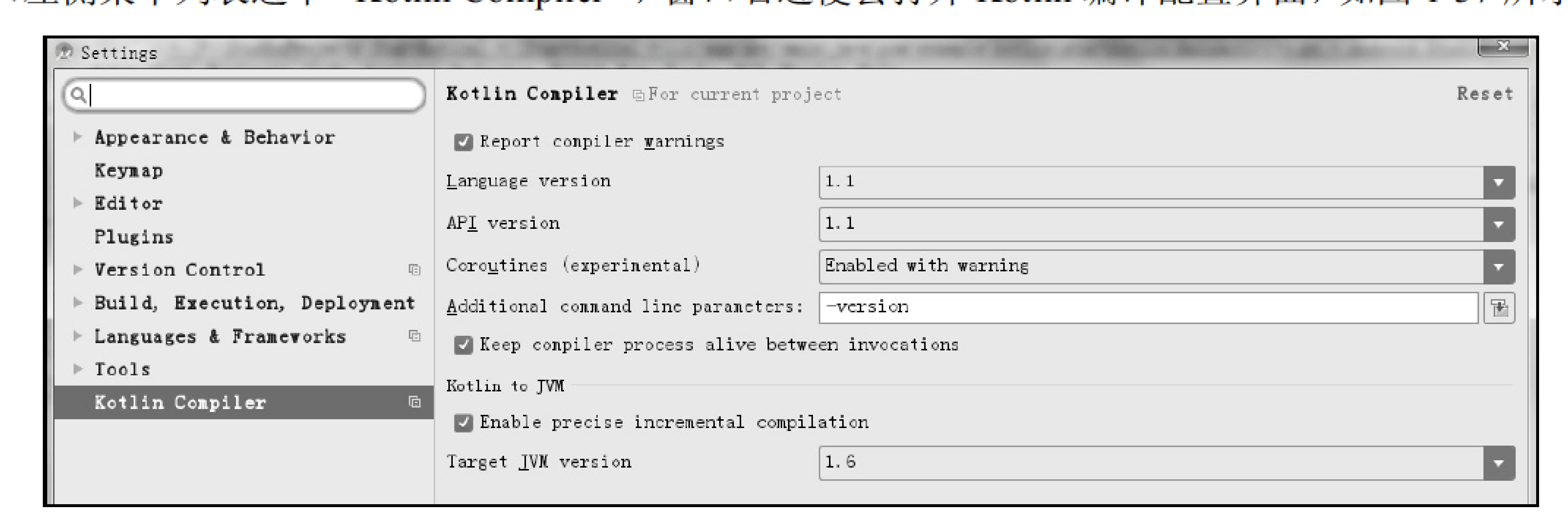


图 1-37　Kotlin 的编译配置界面

可以看到，“Language version”和“API version”目前选的都是 1.1，表示当前 App 工程采用的 Kotlin 编译版本为 1.1。

1.4.2　修改编译配置文件

只看菜单界面上的 Kotlin 编译配置还是不够直截了当，到底这个编译版本是在哪个文件里面配置的呢？先打开工程的编译配置文件 build.gradle 看看，该文件内容如图 1-38 所示。

```
activity_main.xml × | MainActivity.kt × | StartKotlin ×
Gradle files have changed since last project sync. A project sync may be necessary for the IDE

// Top-level build file where you can add configuration options common to all sub-proje

buildscript {
    ext.kotlin_version = '1.1.51'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

图 1-38　工程级别的编译配置文件 build.gradle

图 1-38 所示的 build.gradle 文件内容是下面这样的：

```
buildscript {
    //指定 Kotlin 插件的版本，这里是 Android Studio 3.0.1 默认的 1.1.51
    ext.kotlin_version = '1.1.51'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        //指定 Kotlin 插件的路径
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:
$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}
```

正如图 1-38 框中标记的那样，Kotlin 工程的编译配置文件比 Java 编写的 App 工程多了两处修改，说明如下：

（1）定义了一个外部变量 ext.kotlin_version，其值为 Kotlin 编译版本号“1.1.51”。

（2）指定了 Kotlin 插件的编译路径，即“org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version”。

可是仅仅修改工程级别的 build.gradle 是不够的。再看看模块级别的 build.gradle，该文件内容如图 1-39 和图 1-40 所示，其中图 1-39 所示为文件开头部分的截图，图 1-40 所示为文件末尾 dependencies 块的截图。

```
activity_main.xml × | MainActivity.kt × | StartKotlin × | app ×
Gradle files have changed since last project sync. A project sync may be necessary for

1  apply plugin: 'com.android.application'
2  apply plugin: 'kotlin-android'
3  apply plugin: 'kotlin-android-extensions'
```

图 1-39　模块级别的编译配置文件 build.gradle 开头

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

图 1-40　模块级别的编译配置文件 build.gradle 末尾

注意图 1-39 和图 1-40 框中的部分，这里依然有两个地方与众不同，说明如下：

（1）文件开头增加了两个插件，即'kotlin-android'和'kotlin-android-extensions'，表示该模块会运用 Kotlin 插件功能。补充 Kotlin 插件声明后的文件头部如下所示：

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

（2）文件末尾的 dependencies 块增加了 Kotlin 插件库的编译声明，具体声明语句如下所示：

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
```

综上所述，Kotlin 工程与 Java 编写的 App 工程相比，一共要调整两个 build.gradle 的 4 处编译配置，方能正常支持 Kotlin 代码的编译运行。

1.4.3　Java 代码转 Kotlin 代码

前面介绍了 Kotlin 工程的编译配置说明，如果现在有一个 Java 编码的 App 工程，要如何将其转换为 Kotlin 工程呢？

假设读者目前还没有 Kotlin 基础，那么按照 App 开发的常规流程，先创建一个新模块，依次选择菜单“File”→“New”→“New Module”，然后一路单击“Next”按钮完成模块创建。再按

照“1.4.2 修改编译配置文件”的说明，给这个新模块添加 Kotlin 编译支持。接着打开 MainActivity.java，这个文件的内容再熟悉不过了，就是最简单的几行 Java 代码，如下所示：

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

现在我们要移花接木，把 Java 代码转换为 Kotlin 代码。先选中 MainActivity.java，再到主界面上依次选择菜单“Code”→“Convert Java File to Kotlin File”，菜单位置如图 1-41 所示。

代码转换完毕，原来的 MainActivity.java 变成了 MainActivity.kt，文件内容也变成了如下所示的 Kotlin 代码：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

看起来，这个 Kotlin 的语法与 Java 似曾相识，但又有所不同。若想解释 Kotlin 的详细语法规则，可参见本书第 2 章到第 5 章的语法部分。这里先把 DEMO 跑起来再说，依次选择菜单“Run”→“Run 'hello'”启动应用，正常的话，可在接入的模拟器或者真机上看到“Hello World！”，如图 1-42 所示。

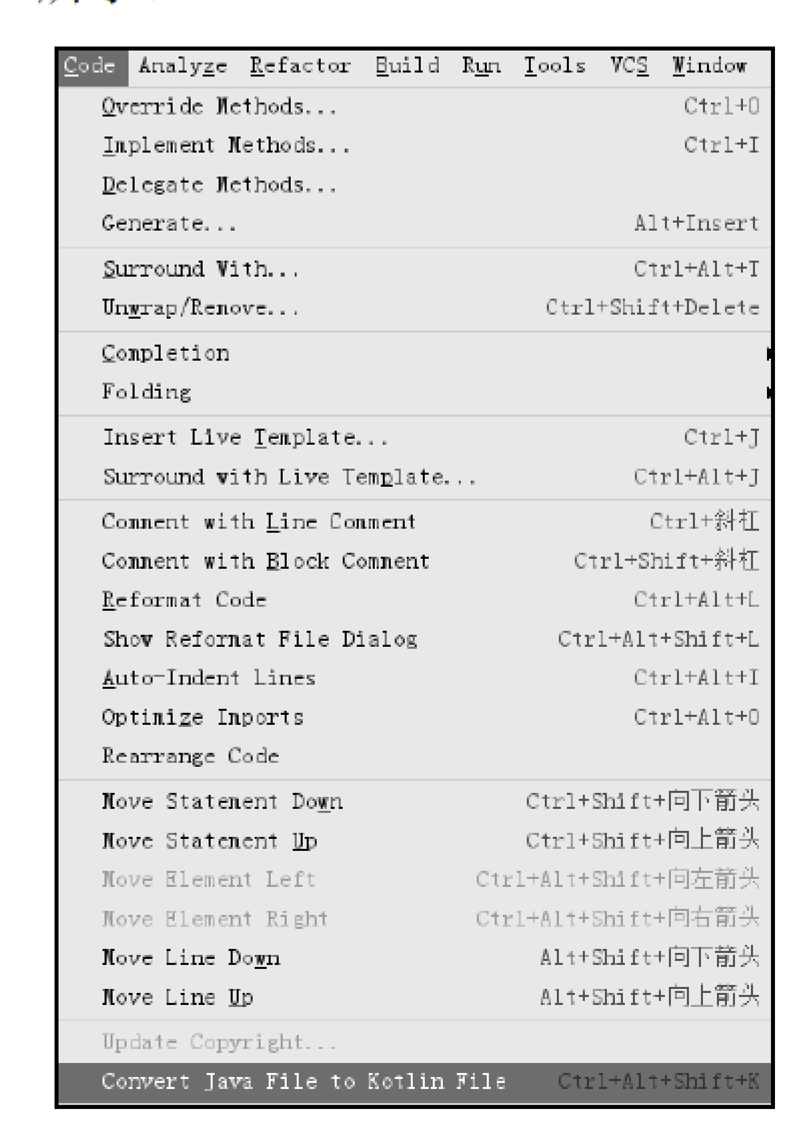


图 1-41　将 Java 代码转换为 Kotlin 代码的菜单位置

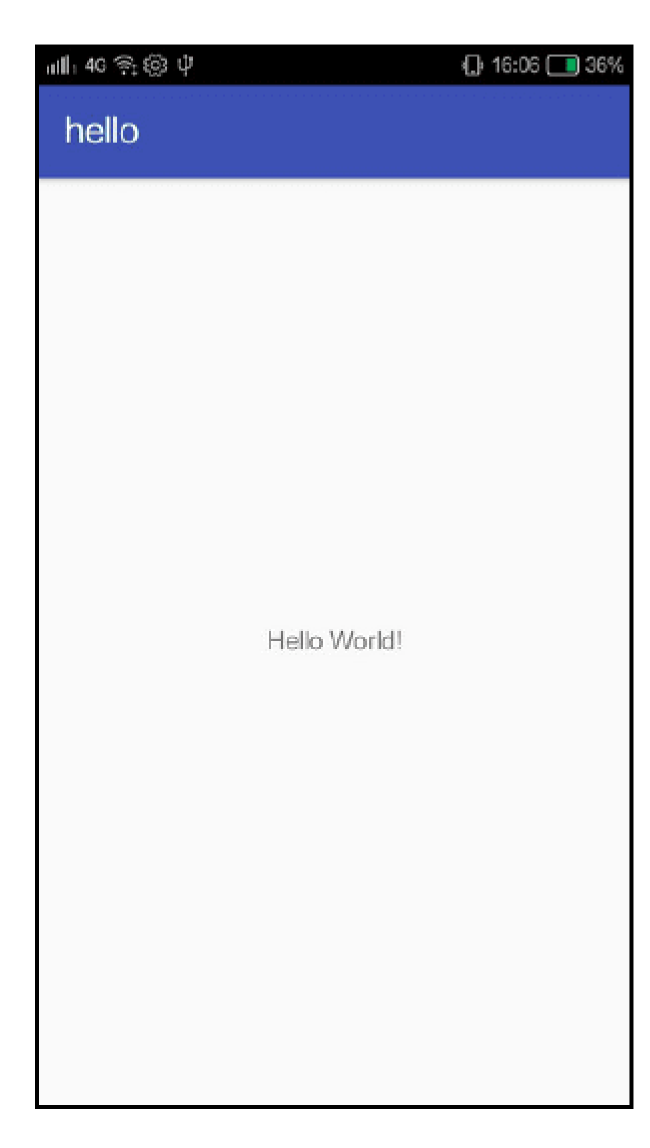


图 1-42　Java 代码转换为 Kotlin 代码之后的测试应用运行界面

怎么样，这可是一个货真价实的用 Kotlin 开发的 App，都说万事开头难，搭建好 Kotlin 的开发环境，只是万里长征的第一步，在下面的章节中，我们将继续学习如何使用 Kotlin 进行 Android 开发。

1.5 Kotlin 相关技术

本节主要介绍 Kotlin 语言在编码过程中运用的一些相关技术，首先对 Kotlin 代码与 Java 代码进行编程效率的比较，然后分别阐述 Kotlin 采用的 Anko 库以及 Lambda 表达式的相关概念以及具体用法。

1.5.1 Kotlin 代码与 Java 代码 PK

前面介绍了如何搭建 Kotlin 的开发环境，可是这个开发环境依然基于 Android Studio，而在 Android Studio 上使用 Java 进行编码本来就是理所应当的，何必还要专门弄个 Kotlin，这个 Kotlin 相比 Java 到底有哪些好处呢？

我们可以把 Kotlin 看作是 Java 的升级版，它不但完全兼容 Java，而且极大地精简了代码语法，从而使开发者专注于业务逻辑的编码，无须在烦琐的代码框架之间周旋。当然，若想充分运用 Kotlin 的优异特性，除了导入 Kotlin 的核心库外，还得导入 Kotlin 的扩展库与 Anko 库。具体到编译配置文件，则要进行以下两处修改：

（1）打开项目的 build.gradle，补充添加 Anko 库的版本号声明，以及 Kotlin 扩展库的路径，完整的编译配置如下所示：

```
    buildscript {
        ext.kotlin_version = "1.2" //指定 Kotlin 的编译版本号
        ext.anko_version = "0.9" //指定 Anko 库的版本号
        repositories {
            google()
            jcenter()
        }
        dependencies {
            classpath 'com.android.tools.build:gradle:3.0.1'
            classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:
$kotlin_version"
            classpath "org.jetbrains.kotlin:kotlin-android-extensions:
$kotlin_version"
        }
    }
```

（2）打开模块的 build.gradle，在文件开头补充添加 Kotlin 的扩展插件，配置添加示例如下：

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

接着在 dependencies 节点下补充添加 Kotlin 与 Anko 插件的编译说明，如下所示：

```
    //Android Studio 3.0 开始使用 implementation，2.*版本使用 compile
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    compile "org.jetbrains.anko:anko-common:$anko_version"
```

编译配置修改完毕，接下来尝试进行简单的 Kotlin 编码，看看 Kotlin 的代码究竟有多么的简练。

首先按照前面“1.2.4 新建 Kotlin 文件”小节的描述，给该模块创建一个名称为 EasyActivity.kt 的 Kotlin 文件，对应的布局文件名则为 activity_easy.xml。然后给布局文件 activity_easy.xml 添加几个 TextView 和 Button 控件，布局比较简单，可参考本书下载资源中的源代码。

接下来是本小节的重点，以前开发者在操纵控件时，都要先通过 findViewById 方法获得控件对象，再调用相关函数设置对象属性。比如现在有一个名为 tv_hello 的 TextView 控件，准备在代码中把 tv_hello 的显示文本改为“你好呀”，如果用 Java 编码，就是下面几行代码：

```
TextView tv_hello = (TextView) findViewById(R.id.tv_hello);
tv_hello.setText("你好呀");
```

如果用 Kotlin 修改文本这个功能，实现会是怎么样的呢？下面就让我们实验一下。首先在 EasyActivity.kt 代码开头补充下面一行：

```
import kotlinx.android.synthetic.main.activity_easy.*
```

这行导入语句的目的是引进 Kotlin 的控件变量自动映射功能，接下来的代码就无须再调用 findViewById 方法，直接把控件 ID 当作控件对象使用即可。比如修改 TextView 的显示文本，采用 Kotlin 编码只要下面一行：

```
tv_hello.setText("你好呀")
```

如此一来，原来的两行代码精简到一行代码，去掉了原先获取控件对象的冗余代码。然而 Kotlin 的便利性并不仅限于此，它对控件甚至都无须调用 set***/get***方法，而允许直接修改/获取控件的属性值，如设置文本这个功能，可以继续简化为下面这行代码：

```
tv_hello.text = "你好呀"
```

进一步简化之后，原代码的“set”与两个括号都被去除，但是新代码反而更容易理解了。

也许有人说，Kotlin 在这里只精简了一行代码，不见得比 Java 有多大优势，那就继续进行其他常见功能的 PK，有道是五局三胜，赢得多才足以服众。上面的第一局为修改控件文本的 PK，结果是 Kotlin 小胜；接下来再设四局 PK，其中第二局为点击监听器的处理。Button 是 Android 的常用按钮控件，代码中经常要处理 Button 控件的点击事件，下面的 Java 代码就是响应 Button 点击的一个例子：

```
    final Button btn_click = (Button) findViewById(R.id.btn_click);
    btn_click.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            btn_click.setText("您点了一下下");
        }
    });
```

其实这个响应功能很简单，仅仅在点击按钮时修改按钮文本而已，可是因为 Java 需要实现点击监听器，所以无奈还得写好几行的匿名类代码。如果使用 Kotlin 实现相同的功能，又是怎样的呢？且看下面的 Kotlin 代码：

```
    btn_click.setOnClickListener { btn_click.text="您点了一下下" }
```

不得了了，Kotlin 只需一行代码就完事，想不到吧，此局 Kotlin 完胜。

第三局换个 Button 控件的长按事件，下面的 Java 代码是响应 Button 长按的一个例子：

```
    final Button btn_click_long = (Button) findViewById(R.id.btn_click_long);
    btn_click_long.setOnLongClickListener(new View.OnLongClickListener() {
        @Override
        public boolean onLongClick(View v) {
            btn_click_long.setText("您长按了一小会");
            return true;
        }
    });
```

可以看到 Java 代码依旧冗长，再看看 Kotlin 代码如何接招：

```
     btn_click_long.setOnLongClickListener { btn_click_long.text="您长按了一小
会"; true }
```

Kotlin 仍旧一行代码搞定，真是叫人刮目相看，此局 Kotlin 依然完胜。

第四局咱不比监听器了，Java 在匿名类这块很吃亏，那来比另一种常用的 Toast 提示功能，该功能的 Java 代码只有一行：

```
    final Button btn_toast = (Button) findViewById(R.id.btn_toast);
    btn_toast.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(EasyJavaActivity.this, "小提示：您点了一下下",
Toast.LENGTH_SHORT).show();
        }
    });
```

上面外层的点击监听器请忽略，正宗的 Toast 代码真的只有一行，且看 Kotlin 怎么拆招：

```
btn_toast.setOnClickListener { toast("小提示：您点了一下下") }
```

哈哈，Kotlin 连同监听器的代码，比 Java 的一行 Toast 代码都要少，此局 Kotlin 继续小胜。

可是为什么 Kotlin 的 toast 函数不区分显示时长呢？原来 toast 方法默认为短时显示，即 Toast.LENGTH_SHORT。这下 Java 方窃喜，虽然我的代码比较长，但是足够灵活呀，想要短一点就 LENGTH_SHORT，想要长一点就 LENGTH_LONG。正好第五局比试 Toast 的长时提示，该功能的 Java 代码也只有一行 Toast：

```
    final Button btn_toast_long = (Button) findViewById(R.id.btn_toast_long);
    btn_toast_long.setOnLongClickListener(new View.OnLongClickListener() {
        @Override
        public boolean onLongClick(View v) {
            Toast.makeText(EasyJavaActivity.this, "长提示：您长按了一小会",
Toast.LENGTH_LONG).show();
            return true;
        }
    });
```

现在 Kotlin 没法调用 toast 函数了吧，Java 洋洋自得总算能够扳回一局，谁料 Kotlin 大喝一声“看我来”：

```
    btn_toast_long.setOnLongClickListener { longToast("长提示：您长按了一小会");
true }
```

真是未曾想到，Kotlin 另外有一个 longToast 招式，仅仅多了 4 个字母而已，于是此局 Kotlin 理应小胜。

五局 PK 下来，Kotlin 大获全胜，Java 溃不成军，直教人长吁短叹“长江后浪推前浪，前浪死在沙滩上”。

1.5.2　Anko 库

Anko 是使用 Kotlin 语言编写的一个 Android 增强库，它用于简化 Android 开发时的 Kotlin 代码，使得开发者只用较少的 Kotlin 代码便能表达完整的编程含义，同时也让 App 代码变得更加简洁易懂。

例如 1.5.1 小节的 toast 和 longToast，这两个函数就在 Anko 库中定义。对于 toast 函数，它在 Anko 库中的原始定义是下面这样的：

```
    fun Context.toast(message: CharSequence) = Toast.makeText(this, message,
Toast.LENGTH_SHORT).show()
```

对于 longToast 函数，它在 Anko 库中的原始定义是下面这样的：

```
    fun Context.longToast(message: CharSequence) = Toast.makeText(this, message,
Toast.LENGTH_LONG).show()
```

注意到 Anko 库的 Toasts.kt 文件是给 Context 类添加了扩展函数 toast 和 longToast，这意味着凡是继承了 Context 的类（包括 Activity、Service 等），均可在类内部代码直接调用 toast 和 longToast 方法实现弹窗提示效果，而不必额外声明工具类对象。

为了正常地使用 toast 和 longToast 函数，要在代码文件头部加上下面两行导入语句：

```
import org.jetbrains.anko.toast
import org.jetbrains.anko.longToast
```

另外，修改项目的 build.gradle，在 buildscript 节点中补充下面一行的 Anko 库版本号定义：

```
ext.anko_version = "0.9" //指定 Anko 库的版本号
```

同时，修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

当然，读者刚看到这里的时候，应该还不具备多少 Kotlin 基础，尚无法理解 Kotlin 的扩展函数与类继承的用法。此处介绍 Anko 库的目的只是告诉读者有这么一种增强库，具体的 Kotlin 语法在后续章节会进行详细和深入的介绍。

1.5.3 Lambda 表达式

Lambda 表达式其实是一个匿名函数，匿名函数指的是：它是一个没有名字的函数，但函数体的内部代码是完整的。可是常规的函数调用都必须指定函数名称，既然匿名函数不存在函数名称，那么其他地方怎样调用它呢？为解答这个问题，先来看看 Android 处理按钮点击事件的 Java 代码片段：

```
btn_click.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        btn_click.setText("您点了一下下");
    }
});
```

上面的代码段摘录于之前的“1.5.1 Kotlin 代码与 Java 代码 PK”小节，显然 Java 的这种写法太过啰嗦，既创建类实例又重写 onClick 函数。其实此处的业务逻辑很简单，仅仅是发生点击事件时修改一下按钮文本就好了，监听代码何必要搞得这么复杂呢？出现该现象的缘由是，Java 是一个纯面向对象的语言，因此它必须按照面向对象的完整写法老老实实地继承类，然后声明类实例，最后重载函数。

Java 设计人员为了保持 Java 代码的严谨性和连贯性，对于上述代码的情况一直只能这么处理。经过多年努力，Java 的设计者终于找到了符合 Java 编程习惯的 Lambda 表达式，也就是简化后的 Java 编码，其中把多余的实例声明与函数重载部分统统去掉，只留下与业务相关的核心代码，从而形成了下面的 Lambda 表达式代码：

```
btn_click.setOnClickListener((View v) -> {
        btn_click.setText("您点了一下下");
});
```

初步精简后的 Lambda 表达式代码只保留了 onClick 函数的输入参数与函数内部代码（二者之间通过“->”连接），连函数名称也被省略掉了。注意到函数内部未使用输入参数 v，所以完全可以把没用的输入参数去掉，于是上面的 Lambda 代码便进一步简化成下面这样：

```
    btn_click.setOnClickListener({
            btn_click.setText("您点了一下下");
    });
```

虽然以上的 Lambda 代码已经够短了，可是仍旧存在改进的空间。仔细观察发现 setOnClickListener 函数在圆括号内部又包了一层花括号，两层括号紧紧贴在一起纯属浪费，因此完全可以把两层括号简写为一层花括号，简写后的 Lambda 代码如下所示：

```
    btn_click.setOnClickListener{
            btn_click.setText("您点了一下下");
    };
```

至此，采取 Lambda 表达式的 Java 点击事件处理代码已经跟下面的 Kotlin 代码很接近了：

```
    btn_click.setOnClickListener { btn_click.text="您点了一下下" }
```

Java 从 1.8 开始支持 Lambda 表达式，如果 Android Studio 采取 JDK 1.7 进行 App 开发，Java 编码是不能使用 Lambda 表达式的。由于 Kotlin 从一诞生就支持 Lambda 表达式，因此并不在乎 JDK 版本是 1.7 还是 1.8，只要采用最新版本的 Kotlin 编译，都能正常使用 Lambda 表达式。

1.6 小　结

本章主要介绍了 Kotlin 开发环境（即 Android Studio）的环境搭建，包括 Kotlin 与 Android 开发的关系、如何安装与配置 Android Studio、如何创建 Kotlin 工程与 Kotlin 文件、如何升级和配置 Android Studio 上的 Kotlin 插件、如何调整 Kotlin 工程的编译配置，并借此初步认识到利用 Kotlin 开发 App 带来的巨大好处。

通过本章的学习，读者应该学会基于 Android Studio 环境的 Kotlin 基本操作步骤，能够正确配置、编译和运行 Kotlin 编码的 App 工程，并具备进一步提高的学习基础。

第 2 章

数据类型

如果把写程序比喻成盖房子，那么各种变量相当于各种建筑材料，建材包括砖头、水泥、沙子等，程序变量也分为不同的数据类型，例如整型数、浮点数、数组、字符串以及更高级的各种容器等。建筑内容有诸如砌砖头、搅拌泥沙等处理操作，数据类型也拥有自己的常见操作，如转换、修改、查询等。下面就依次介绍如何使用 Kotlin 的各种数据类型。

2.1 基本数据类型

每个编程语言都离不开基本的数据类型，包括整型、浮点型、布尔型等，当然 Kotlin 也不例外。虽然基本数据类型的概念是老生常谈，但是 Kotlin 声明基本变量究竟有哪些特别之处呢？本节从基本变量类型开始，逐步探讨这些数据类型的常见用法。

2.1.1 基本类型的变量声明

Kotlin 的基本数据类型跟其他高级语言的分类一样，包括整型、长整型、浮点型、双精度、布尔型、字符型、字符串这几种常见类型，具体的类型名称说明见表 2-1。

表 2-1 Kotlin 与 Java 的基本数据类型对比

基本数据类型名称	Kotlin 的数据类型	Java 的数据类型
整型	Int	int 和 Integer
长整型	Long	long 和 Long
浮点型	Float	float 和 Float
双精度	Double	double 和 Double
布尔型	Boolean	boolean 和 Boolean

（续表）

基本数据类型名称	Kotlin 的数据类型	Java 的数据类型
字符型	Char	char
字符串	String	String

看起来很熟悉是不是，Kotlin 原来这么简单。可是如果你马上敲出变量声明的代码，便会发现编译有问题。比如声明一个最简单的整型变量，按 Java 的写法是下面这样：

```
int i=0;
```

倘若按照 Java 的规则来书写 Kotlin 代码，就是下面这行代码：

```
Int i=0;
```

然而 Android Studio 立即提示编译不通过，刚开始学 Kotlin 便掉到坑里，看来要认真对待 Kotlin，不能这么轻易让它坑蒙拐骗了。正确的 Kotlin 声明变量的代码是下面这样的：

```
var i:Int = 0
```

前面的 var 表示后面是一个变量声明语句，接着是“变量名:变量类型”的格式声明，而不是常见的“变量类型 变量名”这种格式。至于后面的分号，则看该代码行后面是否还有其他语句，如果变量声明完毕直接回车换行，那么后面无须带分号；如果没有回车换行，而是添加其他语句，那么变量声明语句要带上分号。

2.1.2　简单变量之间的转换

Kotlin 变量的另一个重要特点是类型转换，在 Java 开发中，如 int、long、float、double 类型的变量可以直接在变量名前面加上诸如（int）、（long）、（float）、（double）这种表达式进行强制类型转换；对于 int（整型）和 char（字符型）这两种类型，甚至都无须转换类型，直接互相赋值即可。但在 Kotlin 中，不允许通过 Java 的前缀表达式来强制转换类型，只能调用类型转换函数输出其他类型的变量，表 2-2 是常见的几种类型转换函数的说明。

表 2-2　Kotlin 的数据类型转换函数的说明

Kotlin 的数据类型转换函数	转换函数说明
toInt	转换为整型数
toLong	转换为长整型
toFloat	转换为浮点数
toDouble	转换为双精度数
toChar	转换为字符
toString	转换为字符串

接下来通过实际代码观察一下类型转换的过程，测试用到的类型转换的 Kotlin 代码片段如下所示：

```
    val origin:Float = 65.0f
    tv_origin.text = origin.toString()
    var int:Int
    btn_int.setOnClickListener { int=origin.toInt();
tv_convert.text=int.toString() }
    var long:Long
    btn_long.setOnClickListener { long=origin.toLong();
tv_convert.text=long.toString() }
    var float:Float
    btn_float.setOnClickListener { float=origin.toDouble().toFloat();
tv_convert.text=float.toString() }
    var double:Double
    btn_double.setOnClickListener { double=origin.toDouble();
tv_convert.text=double.toString() }
    var boolean:Boolean
    btn_boolean.setOnClickListener { boolean=origin.isNaN();
tv_convert.text=boolean.toString() }
    var char:Char
    btn_char.setOnClickListener { char=origin.toChar();
tv_convert.text=char.toString() }
```

各种类型转换的操作结果如图 2-1～图 2-3 所示，其中图 2-1 展示转换为整型的界面效果，图 2-2 展示转换为双精度的界面效果，图 2-3 展示转换为字符型的界面效果。



图 2-1　浮点型转换为整型



图 2-2　浮点型转换为双精度



图 2-3　浮点型转换为字符型

注意到上述类型转换代码的第一行变量声明语句以 val 开头，而其余的变量声明语句均以 var 开头，这是为什么呢？其实 val 和 var 的区别在于，前者修饰过的变量只能在第一次声明时赋值，后续不能再赋值；而后者修饰过的变量在任何时候都允许赋值。方便记忆的话，可以把 val 看作是 Java 里的 final 关键字；至于 var，Java 里面没有对应的关键字，就当它是例行公事好了。

2.2　数　　组

2.1 节介绍了基本数据类型在 Kotlin 中的用法，不过这只针对单个变量，如果要求把一组相同类型的变量排列起来，形成一个变量数组，那又该如何声明和操作呢？本节就来谈谈 Kotlin 对数组的常见用法。

2.2.1 数组变量的声明

在 Java 中声明数组跟在 C 语言中声明是一样的，以整型数组为例，声明数组并加以初始化的语句如下所示：

```
int[] int_array = new int[] {1, 2, 3};
```

其他基本类型的数组声明与之类似，只要把 int 替换为 long、float、double、boolean、char 其中之一即可。但在 Kotlin 中，声明并初始化一个整型数组的语句是下面这样的：

```
var int_array:IntArray = intArrayOf(1, 2, 3)
```

两相对比，对于整型数组的声明，Kotlin 与 Java 之间有以下区别：

（1）Kotlin 另外提供了新的整型数组类型，即 IntArray。

（2）分配一个常量数组，Kotlin 调用的是 intArrayOf 方法，并不使用 new 关键字。

推而广之，其他基本类型的数组也各有自己的数组类型，以及对应分配常量数组的初始化方法，详细的对应关系说明见表 2-3。

表 2-3　Kotlin 基本数据类型名称及其初始化方法对应关系

Kotlin 的基本数组类型	数组类型的名称	数组类型的初始化方法
整型数组	IntArray	intArrayOf
长整型数组	LongArray	longArrayOf
浮点数组	FloatArray	floatArrayOf
双精度数组	DoubleArray	doubleArrayOf
布尔型数组	BooleanArray	booleanArrayOf
字符数组	CharArray	charArrayOf

下面是这些基本类型数组的初始化代码例子：

```
var long_array:LongArray = longArrayOf(1, 2, 3)
var float_array:FloatArray = floatArrayOf(1.0f, 2.0f, 3.0f)
var double_array:DoubleArray = doubleArrayOf(1.0, 2.0, 3.0)
var boolean_array:BooleanArray = booleanArrayOf(true, false, true)
var char_array:CharArray = charArrayOf('a', 'b', 'c')
```

不知读者有没有注意到，上面的 Kotlin 数组类型不包括字符串数组，而 Java 是允许使用字符串数组的，声明字符串数组的 Java 代码示例如下：

```
String[] string_array = new String[] {"How", "Are", "You"};
```

但在 Kotlin 这里，并不存在名为 StringArray 的数组类型，因为 String 是一种特殊的基本数据类型。要想在 Kotlin 中声明字符串数组，得使用 Array<String>类型，也就是把“String”用尖括号包起来。同时，分配字符串数组的方法也相应变成了 arrayOf，下面是声明字符串数组的 Kotlin 代码：

```
var string_array:Array<String> = arrayOf("How", "Are", "You")
```

这种字符串数组的声明方式是不是很熟悉？看起来就跟 Java 里面的 ArrayList 用法差不多，都是在尖括号中间加入数据结构的类型。同理，其他类型的数组变量也能通过“Array<数据类型>”的方式来声明，像前面介绍的整型数组，其实可以使用类型 Array<Int>，以此类推，改造之后的各类型数组变量的声明代码如下所示：

```
var int_array:Array<Int> = arrayOf(1, 2, 3)
var long_array:Array<Long> = arrayOf(1, 2, 3)
var float_array:Array<Float> = arrayOf(1.0f, 2.0f, 3.0f)
var double_array:Array<Double> = arrayOf(1.0, 2.0, 3.0)
var boolean_array:Array<Boolean> = arrayOf(true, false, true)
var char_array:Array<Char> = arrayOf('a', 'b', 'c')
```

2.2.2 数组元素的操作

现在声明数组和对数组初始化的代码都有了，接下来还需要对数组做进一步的处理，常见的处理包括获取数组长度、获取指定位置的数组元素等，这些操作在 Kotlin 与 Java 之间的区别包括：

（1）对于如何获取数组长度，Java 使用.length，而 Kotlin 使用.size。

（2）对于如何获取指定位置的数组元素，Java 通过方括号加下标来获取，比如“int_array[0]”指的是得到该数组的第一个元素；Kotlin 也能通过方括号加下标来获取指定元素，不过 Kotlin 还拥有 get 和 set 两个方法，通过 get 方法获取元素值，通过 set 方法修改元素值，看起来就像在操作 ArrayList 队列。

下面是 Kotlin 操作字符串数组的示例代码：

```
//声明字符串数组
var string_array:Array<String> = arrayOf("How", "Are", "You")
btn_string.setOnClickListener {
    var str:String = ""
    var i:Int = 0
    while (i<string_array.size) {
        str = str + string_array[i] + ", "
        //数组元素可以通过下标访问，也可通过 get 方法访问
        //str = str + string_array.get(i) + ", "
        i++
    }
    tv_item_list.text = str
}
```

上述代码的演示效果如图 2-4 所示，可以看到字符串数组内部的各元素都被逗号分隔开了。

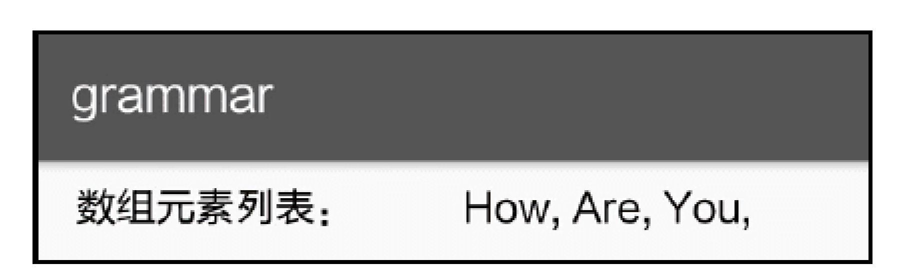


图 2-4 Kotlin 操作字符串数组的演示界面

2.3 字 符 串

2.2 节介绍了数组的声明和操作，其中包括字符串数组的用法。注意到 Kotlin 的字符串类型名称跟 Java 一样都叫 String，那么字符串在 Kotlin 和 Java 中的用法有哪些差异呢？这便是本节所要阐述的内容了。

2.3.1 字符串与基本类型的转换

首先要说明的是字符串类型与基本变量类型之间的转换方式，在前面的“2.1.2 简单变量之间的转换”中，提到基本数据类型的变量可以通过 toString 方法转换为字符串类型。反过来，字符串类型又该如何转换为基本变量类型呢？表 2-4 展示使用 Kotlin 和 Java 编码将字符串转换为基本数据类型的对照方式说明。

表 2-4 字符串转换为其他数据类型的 Kotlin 与 Java 方式对比

字符串转换目标	Kotlin 的转换方式	Java 的转换方式
字符串转整型	字符串变量的 toInt 方法	Integer.parseInt（字符串变量）
字符串转长整型	字符串变量的 toLong 方法	Long.parseLong（字符串变量）
字符串转浮点数	字符串变量的 toFloat 方法	Float.parseFloat（字符串变量）
字符串转双精度数	字符串变量的 toDouble 方法	Double.parseDouble（字符串变量）
字符串转布尔型	字符串变量的 toBoolean 方法	Boolean.parseBoolean（字符串变量）
字符串转字符数组	字符串变量的 toCharArray 方法	字符串变量的 toCharArray 方法

就表 2-4 的转换情况来看，Java 的实现方式比较烦琐，既需要其他类型的类名，又需要该类型的转换方法。而在 Kotlin 这边，转换类型相对简单，并且与基本数据类型之间的转换形式保持一致，即都是采取“to***()”的形式。显而易见，Kotlin 对字符串的类型转换方式更友好，也更方便记忆。

2.3.2 字符串的常用方法

当然，转换类型只是字符串的基本用法，还有更多处理字符串的其他用法，比如查找子串、替换子串、截取指定位置的子串、按特定字符分隔子串等，在这方面 Kotlin 基本兼容 Java 的相关方法。对于查找子串的操作，二者都调用 indexOf 方法；对于截取指定位置子串的操作，二者都调用 substring 方法；对于替换子串的操作，二者都调用 replace 方法；对于按特定字符分隔子串的操作，二者都调用 split 方法。

下面是 Kotlin 使用 indexOf 和 substring 方法进行子串查找和截取字符串的代码例子：

```
    //截取小数点之前的字符串，即取整操作
    val origin:String = tv_origin.text.toString()
```

```
var origin_trim:String = origin
if (origin_trim.indexOf('.') > 0) {
    origin_trim = origin_trim.substring(0, origin_trim.indexOf('.'))
}
```

在这些字符串处理方法里面，唯一区别是 split 方法的返回值，在 Java 中，split 方法返回的是 String 数组，即 String[]；但在 Kotlin 中，split 方法返回的是 String 队列，即 List<String>。下面是 Kotlin 使用 split 方法的示例代码：

```
//根据点号将源串分割为字符串队列，并将分割结果显示在界面上
btn_split.setOnClickListener {
    var strList:List<String> = origin.split(".")
    var strResult:String = ""
    for (item in strList) {
        strResult = strResult + item + ", "
    }
    tv_convert.text = strResult
}
```

分割字符串的界面效果如图 2-5 所示，可以看到源字符串里面的点号都被替换为逗号，字符串末尾也多了一个逗号。

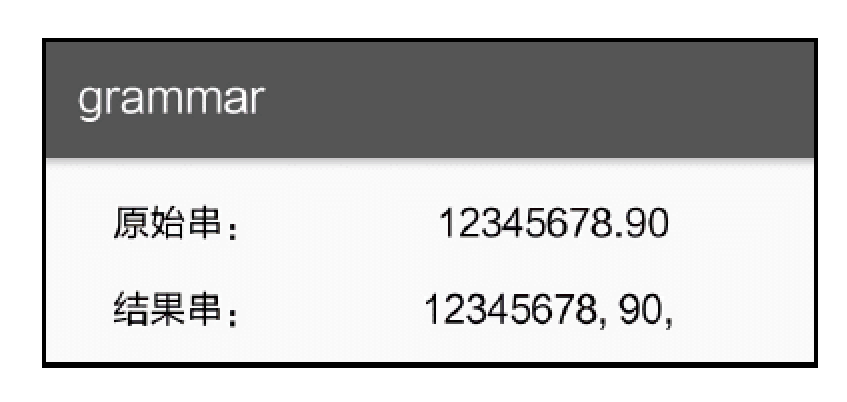


图 2-5　Kotlin 调用字符串方法的演示界面

若想获取字符串某个位置的字符，这个看似简单的需求，采取 Java 实现时却有点烦琐，因为只能调用 substring 方法去截取指定位置的字符串，具体的Java代码如下所示：

```
String result = origin.substring(number, number+1);
tv_convert.setText(result);
```

通过 Kotlin 实现上述需求就简单多了，因为 Kotlin 允许直接通过下标访问字符串指定位置的字符，下面是访问字符串指定位置的 Kotlin 代码例子：

```
tv_convert.text = origin[number].toString()
```

同时，Kotlin 也支持字符串变量通过 get 方法获取指定位置上的字符，代码如下：

```
tv_convert.text = origin.get(number).toString()
```

如此一来，Kotlin 的字符串定位代码不但更加精炼，而且可读性也增强了。

2.3.3　字符串模板及其拼接

Kotlin 对字符串带来的便利并不限于此，举个例子，若 Java 把几个变量拼接成字符串，则要么用加号强行拼接，要么用 String.format 函数进行格式化。可是前者的拼接加号时常会跟数值相加的加号混淆；而后者的格式化还得开发者死记硬背，如%d、%f、%s、%c、%b 等格式转换符，实在令人头痛。对于字符串格式化这个痛点，Kotlin 恰如其分地进行了优化，何必引入这些麻烦的格式转换符呢？直接在字符串中加入“$变量名”即可表示此处引用该变量的值，岂不妙哉！

心动不如行动，赶紧动起手来，看看 Kotlin 如何格式化字符串，先来看一个示例代码：

```
        btn_format.setOnClickListener { tv_convert.text = "字符串值为 $origin" }
```

这里要注意，符号$后面跟变量名，系统会自动匹配最长的变量名。比如下面这行代码，打印出来的是变量 origin_trim 的值，而不是 origin 的值：

```
        btn_format.setOnClickListener { tv_convert.text = "字符串值为
$origin_trim" }
```

另外，有可能变量会先进行运算，再把运算结果拼接到字符串中。此时，需要用大括号把运算表达式给括起来，具体代码如下所示：

```
        btn_length.setOnClickListener { tv_convert.text = "字符串长度为
${origin.length}" }
```

在上述的 Kotlin 格式化代码中，美元符号$属于特殊字符，因此不能直接打印它，必须经过转义才可以打印。转义的办法是使用“${'***'}”表达式，该表达式外层的“${''}”为转义声明，内层的“***”为需要原样输出的字符串，所以通过表达式“${'$'}”即可打印一个美元符号，示例代码如下：

```
        btn_dollar.setOnClickListener { tv_convert.text = "美元金额为
${'$'}$origin" }
```

如果只是对单个美元符号做转义，也可直接在符号$前面加个反斜杆，即变成“\$”，修改后的代码如下所示：

```
        btn_dollar.setOnClickListener { tv_convert.text = "美元金额为
\$$origin" }
```

然而一个反斜杆仅仅对一个字符进行转义，倘若要对一个字符串做转义，也就是把某个字符串的所有字符原样输出，那么只能采用形如“${'***'}”的表达式，该表达式利用单引号把待转义的字符串包起来，好处是能够保留该字符串内部的所有特殊字符。

2.4　容　　器

Kotlin 号称全面兼容 Java，于是 Java 的容器类仍可在 Kotlin 中正常使用，包括大家熟悉的队列 ArrayList、映射 HashMap 等。不过 Kotlin 作为一门全新的语言，肯定要有自己的容器类，不然哪天 Java 跟 Kotlin 划清界限，那麻烦就大了。本节就对 Kotlin 的几种容器类进行详细的说明。

2.4.1　容器的基本操作

与 Java 类似，Kotlin 也拥有三类基本的容器，分别是集合 Set、队列 List、映射 Map，然后每类容器又分作只读与可变两种类型，这是为了判断该容器能否进行增、删、改等变更操作。Kotlin 对变量的修改操作很慎重，每个变量在定义的时候就必须指定能否修改，比如添加 val 修饰表示该

变量不可修改，添加 var 修饰表示该变量允许修改。至于容器则默认为只读容器，如果需要允许修改该容器变量，就需要加上 Mutable 前缀形成新的容器，比如 MutableSet 表示可变集合，MutableList 表示可变队列，MutableMap 表示可变映射，只有可变的容器才能够对其内部元素进行增、删、改操作。

既然集合 Set、队列 List、映射 Map 三者都属于容器，那么它们必定拥有相同的容器方法，这些公共方法具体说明如下。

- isEmpty：判断该容器是否为空。
- isNotEmpty：判断该容器是否非空。
- clear：清空该容器。
- contains：判断该容器是否包含指定元素。
- iterator：获取该容器的迭代器。
- count：获取该容器包含的元素个数，也可通过 size 属性获得元素数量。

另外，Kotlin 允许在声明容器变量时就进行初始赋值，如同对数组变量进行初始化那样。而 Java 的容器类是无法同时声明并初始化的，由此可见 Kotlin 的这点特性给开发者带来很大便利。下面是一个初始化 List 队列的 Kotlin 代码例子：

```
        val satellites:List<String> = listOf("水星", "金星", "地球", "火星",
"木星", "土星")
```

当然，不同容器的初始化方法有所区别，各种容器与其初始化方法的对应关系见表 2-5。

表 2-5 Kotlin 的容器及其初始化方法的对应关系

Kotlin 的容器	容器名称	容器的初始化方法
只读集合	Set	setOf
可变集合	MutableSet	mutableSetOf
只读队列	List	listOf
可变队列	MutableList	mutableListOf
只读映射	Map	mapOf
可变映射	MutableMap	mutableMapOf

以上介绍了 Kotlin 容器的基本用法，更具体的增、删、改、查等操作则有所不同，接下来分别说明这三类 6 种容器的详细使用。

2.4.2 集合 Set/MutableSet

集合是一种最简单的容器，它具有以下特性：

（1）容器内部的元素不按顺序排列，因此无法按照下标进行访问。

（2）容器内部的元素存在唯一性，通过哈希值校验是否存在相同的元素，若存在，则将其覆盖。

因为 Set 是只读集合，初始化赋值后便不可更改，所以元素变更的方法只适用于可变集合

MutableSet，但 MutableSet 的变更操作尚有以下限制：

（1）MutableSet 的 add 方法仅仅往集合中添加元素，由于集合是无序的，因此不知道添加的具体位置。

（2）MutableSet 没有修改元素值的方法，一个元素一旦被添加，就不可被修改。

（3）MutableSet 的 remove 方法用于删除指定元素，但无法删除某个位置的元素，这是因为集合内的元素不是按顺序排列的。

对于集合的遍历操作，Kotlin 提供了好几种方式，有熟悉的 for-in 循环、迭代器遍历，还有新面孔 forEach 遍历，这三种集合遍历的用法说明如下。

1. for-in 循环

与 Java 类似，通过 for 语句加上 in 条件即可轻轻松松依次取出集合中的所有元素。下面是运用 for-in 循环的代码例子：

```
    val goodsMutSet:Set<String> = setOf("iPhone8", "Mate10", "小米 6", "OPPO R11",
"vivo X9S", "魅族 Pro6S")
    btn_set_for.setOnClickListener {
        var desc = ""
        //使用 for-in 语句循环取出集合中的每条记录
        for (item in goodsMutSet) {
            desc = "${desc}名称：${item}\n"
        }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

上述代码对应的界面效果如图 2-6 所示，可见初始化时输入的 6 部手机都通过遍历操作打印了出来。



图 2-6 Kotlin 集合的遍历结果

2. 迭代器遍历

迭代器与指针的概念有点接近，它自身并非具体的元素，而是指向元素的存放地址，所以迭代器遍历其实是遍历所有元素的地址。迭代器通过 hasNext 方法判断是否还存在下一个节点，如果不存在下一节点，就表示已经遍历完毕，它通过 next 方法获得下一个节点的元素，同时迭代器自身改为指向该元素的地址。下面是运用迭代器遍历的代码例子：

```
    btn_set_iterator.setOnClickListener {
        var desc = ""
        val iterator = goodsMutSet.iterator()
```

```
        //如果迭代器还存在下一个节点，就继续取出下一个节点的记录
        while (iterator.hasNext()) {
            val item = iterator.next()
            desc = "${desc}名称：${item}\n"
        }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

3. forEach 遍历

无论是 for-in 循环还是迭代器遍历，其实都脱胎于 Java 已有的容器遍历操作，代码书写上不够精炼。为了将代码精简到极致，Kotlin 给容器创造了 forEach 方法，明确指定该方法就是要依次遍历容器内部的元素。forEach 方法在编码时采用匿名函数的形式，内部使用 it 代表每个元素，下面是运用 forEach 遍历的代码例子：

```
    btn_set_foreach.setOnClickListener {
        var desc = ""
        //forEach 内部使用 it 指代每条记录
        goodsMutSet.forEach { desc = "${desc}名称：${it}\n" }
        tv_set_result.text = "手机畅销榜包含以下${goodsMutSet.size}款手机：\n$desc"
    }
```

结合以上有关 Set/MutableSet 的用法说明，可以发现集合在实战中存在诸多不足，主要包括以下几点：

（1）集合不允许修改内部元素的值。
（2）集合无法删除指定位置的元素。
（3）不能通过下标获取指定位置的元素。

鉴于集合的以上缺点难以克服，故而实际开发基本用不到集合，大多数场合用的是它的两个兄弟——队列和映射。

2.4.3 队列 List/MutableList

队列是一种元素之间按照顺序排列的容器，它与集合的最大区别在于多了次序管理。不要小看这个有序性，正因为队列建立了秩序规则，所以它比集合多提供了如下功能（注意，凡是涉及增、删、改的，都必须由 MutableList 来完成）：

（1）队列能够通过 get 方法获取指定位置的元素，也可以直接通过下标获得该位置的元素。
（2）MutableList 的 add 方法每次都是把元素添加到队列末尾，也可指定添加的位置。
（3）MutableList 的 set 方法允许替换或者修改指定位置的元素。
（4）MutableList 的 removeAt 方法允许删除指定位置的元素。
（5）队列除了拥有跟集合一样的三种遍历方式（for-in 循环、迭代器遍历、forEach 遍历）外，还多了一种按元素下标循环遍历的方式，具体的下标遍历代码例子如下：

```
    val goodsMutList:List<String> = listOf("iPhone8", "Mate10", "小米 6", "OPPO R11",
"vivo X9S", "魅族 Pro6S")
    btn_for_index.setOnClickListener {
        var desc = ""
        //indices 是队列的下标数组。如果队列大小为 10，下标数组的取值就为 0～9
        for (i in goodsMutList.indices) {
            val item = goodsMutList[i]
            desc = "${desc}名称：${item}\n"
        }
        tv_list_result.text = "手机畅销榜包含以下${goodsMutList.size}款手机：\n$desc"
    }
```

上面按下标遍历队列的代码对应的运行界面如图 2-7 所示，可见队列中保存的 6 部手机都通过遍历显示了出来。



图 2-7　Kotlin 队列的遍历结果

（6）MutableList 提供了 sort 系列方法用于给队列中的元素重新排序，其中 sortBy 方法表示按照指定条件升序排列，sortByDescending 方法表示按照指定条件降序排列。下面是一个给队列排序的代码例子（含升序和降序）：

```
    var sortAsc = true
    btn_sort_by.setOnClickListener {
        if (sortAsc) {
            //sortBy 表示升序排列，后面跟的是排序条件
            goodsMutList.sortBy { it.length }
        } else {
            //sortByDescending 表示降序排列，后面跟的是排序条件
            goodsMutList.sortByDescending { it.length }
        }
        var desc = ""
        for (item in goodsMutList) {
            desc = "${desc}名称：${item}\n"
        }
        tv_list_result.text = "手机畅销榜已按照${if (sortAsc) "升序" else "降序"}重
新排列：\n$desc"
        sortAsc = !sortAsc
    }
```

队列进行排序操作后的演示效果如图2-8和图2-9所示，其中图2-8展示按升序排列后的手机信息界面，图2-9展示按降序排列后的手机信息界面。

grammar
手机畅销榜已按照长度升序重新排列：
名称：小米6
名称：Mate10
名称：iPhone8
名称：魅族Pro6S
名称：OPPO R11
名称：vivo X9S

图2-8 队列进行升序排列后的结果界面

grammar
手机畅销榜已按照长度降序重新排列：
名称：OPPO R11
名称：vivo X9S
名称：iPhone8
名称：魅族Pro6S
名称：Mate10
名称：小米6

图2-9 队列进行降序排列后的结果界面

2.4.4 映射Map/MutableMap

映射内部保存的是一组键值对（Key-Value），也就是说，每个元素都由两部分构成，第一部分是元素的键，相当于元素的名字；第二部分是元素的值，存放着元素的详细信息。元素的键与值是一一对应的关系，相同键名指向的键值是唯一的，所以映射中每个元素的键名各不相同，这个特性使得映射的变更操作与队列存在以下不同之处（注意，增、删操作必须由MutableMap来完成）：

（1）映射的containsKey方法判断是否存在指定键名的元素，containsValue方法判断是否存在指定键值的元素。

（2）MutableMap的put方法不单单是添加元素，而是智能的数据存储。每次调用put方法时，映射会先根据键名寻找同名元素，如果找不到就添加新元素，如果找得到就用新元素替换旧元素。

（3）MutableMap的remove方法是通过键名来删除元素的。

（4）调用mapOf和mutableMapOf方法初始化映射时，有两种方式可以表达单个键值对元素，其一是采取“键名 to 键值”的形式，其二是采取Pair配对方式，形如“Pair(键名, 键值)”。下面是这两种初始化方式的代码例子：

```
    //to方式初始化映射
    var goodsMap: Map<String, String> = mapOf("苹果" to "iPhone8", "华为" to "Mate10", "小米" to "小米6", "欧珀" to "OPPO R11", "步步高" to "vivo X9S", "魅族" to "魅族Pro6S")
    //Pair方式初始化映射
    var goodsMutMap: MutableMap<String, String> = mutableMapOf(Pair("苹果", "iPhone8"), Pair("华为", "Mate10"), Pair("小米", "小米6"), Pair("欧珀", "OPPO R11"), Pair("步步高", "vivo X9S"), Pair("魅族", "魅族Pro6S"))
```

映射的遍历与集合类似，也有for-in循环、迭代器遍历、forEach遍历三种遍历手段。但是由于映射的元素是一个键值对，因此它的遍历方式与集合稍有不同，详述如下：

1. for-in循环

for-in语句取出来的是映射的元素键值对，若要获取该元素的键名，还需访问元素的key属性；若要获取该元素的键值，还需访问元素的value属性。下面是在映射中运用for-in循环的代码例子：

```
btn_map_for.setOnClickListener {
    var desc = ""
    //使用 for-in 语句循环取出映射中的每条记录
    for (item in goodsMutMap) {
        //item.key 表示该配对的键，即厂家名称；item.value 表示该配对的值，即手机名称
        desc = "${desc}厂家：${item.key}，名称：${item.value}\n"
    }
    tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
}
```

上述通过 for-in 语句遍历映射的运行效果如图 2-10 所示，可见映射内部每个元素的键名与键值都被获取到了。



图 2-10　Kotlin 映射的遍历结果

2. 迭代器遍历

映射的迭代器通过 next 函数得到下一个元素，接着需访问该元素的 key 属性获取键名，访问该元素的 value 属性获取键值。下面是在映射中运用迭代器遍历的代码例子：

```
btn_map_iterator.setOnClickListener {
    var desc = ""
    val iterator = goodsMutMap.iterator()
    //如果迭代器还存在下一个节点，就继续取出下一个节点的记录
    while (iterator.hasNext()) {
        val item = iterator.next()
        desc = "${desc}厂家：${item.key}，名称：${item.value}\n"
    }
    tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
}
```

3. forEach 遍历

映射的 forEach 方法内部依旧采用匿名函数的形式，同时把元素的 key 和 value 作为匿名函数的输入参数。不过映射的 forEach 函数需要 API 24 及以上版本支持，开发时注意修改编译配置。下面是在映射中运用 forEach 遍历的代码例子：

```
btn_map_foreach.setOnClickListener {
    var desc = ""
    //映射的 forEach 函数需要 API 24 及以上版本支持
    //forEach 内部使用 key 指代每条记录的键，使用 value 指代每条记录的值
```

```
        goodsMap.forEach { key, value -> desc = "${desc}厂家：${key}，名称：
${value}\n" }
        tv_map_result.text = "手机畅销榜包含以下${goodsMutMap.size}款手机：\n$desc"
        //tv_map_result.text = "Map 的 forEach 函数需要 API 24 及以上版本支持"
    }
```

2.5 小　　结

本章介绍了 Kotlin 开发涉及的几种常见数据类型，包括以整型、浮点型、布尔型、字符型为代表的基本数据类型，还有这些基本数据类型数组的概念和运用，以及字符串的各种常见用法，最后是几种容器的常见操作方式。

通过本章的学习，读者应能掌握以下技能：

（1）学会 Kotlin 对基本数据类型的变量定义以及变量之间的类型转换。

（2）学会 Kotlin 对基本类型数组的声明方式以及数组变量的常见用法。

（3）学会 Kotlin 对字符串的各种处理操作以及字符串模板的书写格式。

（4）学会 Kotlin 对容器的声明方式及其增、删、改、查操作，包括集合、队列、映射三种基本容器。

第 3 章

控制语句

第 2 章在介绍字符串和容器时，示例代码多次用到 if 和 for 语句，表面上看，Kotlin 对控制语句的处理与 Java 很像，但实际上，Kotlin 在这方面做了不少改进，所以本章针对条件、循环、空值判断、等式判断等控制语句进行详细的说明。

3.1 条件分支

条件分支是最简单的控制语句，主要包括非此即彼的两路分支以及如数家珍的多路分支，下面一起来看看 Kotlin 给条件分支带来了哪些变化。

3.1.1 简单分支

说起条件判断，最简单的莫过于人尽皆知的 if...else...了，这条语句从 C 语言延续到 Java，再进化到 Kotlin，基本用法仍是一样的，看看下面的示例代码就知道了：

```
var is_odd:Boolean = true;
tv_puzzle.text = "凉风有信，秋月无边。打二字"
btn_if_simple.setOnClickListener {
    if (is_odd == true) {
        tv_answer.text = "凉风有信的谜底是“讽”"
    } else {
        tv_answer.text = "秋月无边的谜底是“二”"
    }
    is_odd = !is_odd
}
```

以上代码的作用是，奇数次点击按钮时，界面展示凉风有信的谜底；偶数次点击按钮时，界面展示秋月无边的谜底。看似不能再简单的判断语句，谁能料到 Kotlin 也要加以简化？注意到两个谜底都是显示在控件 tv_answer 上，所以两个分支都出现了“tv_answer.text = ***”的语句。Kotin 在这里要做的优化便是允许分支语句返回字符串，从而在条件语句外层直接对 tv_answer 赋值，优化后的代码如下所示：

```
    btn_if_simple.setOnClickListener {
        tv_answer.text = if (is_odd == true) {
            "凉风有信的谜底是“讽”"
        } else {
            "秋月无边的谜底是“二”"
        }
        is_odd = !is_odd
    }
```

优化后的代码还可以进一步改进，因为每个分支内部只有一个字符串返回值，所以不妨去掉大括号，并且把整个条件语句精简到一行代码，就像下面这样：

```
    btn_if_value.setOnClickListener {
        tv_answer.text = if (is_odd==true) "凉风有信的谜底是“讽”" else "秋月无边的
谜底是“二”"
        is_odd = !is_odd
    }
```

精简了的代码是不是似曾相识？仿佛脱胎于 Java 的三元运算符“变量名=条件语句?取值 A:取值 B”。可是 Kotlin 并不提供这个三元运算符，因为使用上述的 if/else 语句已经实现了同样的功能，所以多余的三元运算符就被取消了。

以上一共实现了三种写法的 Kotlin 条件代码，这三种写法的运行效果完全一模一样，具体结果如图 3-1 和图 3-2 所示，其中图 3-1 所示为奇数次点击按钮的效果图，此时界面显示凉风有信的谜底，图 3-2 所示为偶数次点击按钮的效果图，此时界面显示秋月无边的谜底。

grammar
谜题： 凉风有信，秋月无边。打二字
谜底： 凉风有信的谜底是“讽”

图 3-1 进入奇数次点击分支的界面

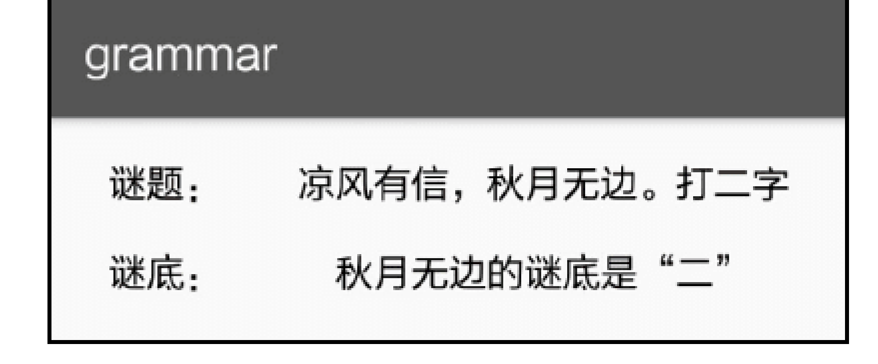


图 3-2 进入偶数次点击分支的界面

3.1.2 多路分支

三元运算符既然已经被取消，一旁的 switch/case 瑟瑟发抖，嘴里嘟囔道：“俺这个多路分支还在不在呀？”看官莫急，虽然 Kotlin 对 if/else 进行了增强，但是仍无法取代多路分支；相反的是，Kotlin 对多路分支的功能做了大幅扩充，当然由于原来的 switch/case 机制存在局限，故而 Kotlin 推出新的关键字，即用 when/else 来处理多路分支的条件判断。

下面来段多路分支用到的 when/else 语句的具体代码例子：

```
var count:Int = 0
btn_when_simple.setOnClickListener {
    when (count) {
        0 -> tv_answer.text = "凉风有信的谜底是“讽”"
        1 -> tv_answer.text = "秋月无边的谜底是“二”"
        //if 语句可以没有 else，但是 when 语句必须带上 else
        else -> tv_answer.text = "好诗，这真是一首好诗"
    }
    count = (count+1) % 3
}
```

从以上代码可以看出 when/else 与 switch/case 有以下几点区别：

（1）关键字 switch 被 when 取代。

（2）判断语句“case 常量值:”被新语句“常量值 ->”取代。

（3）每个分支后面的 break 语句取消了，因为 Kotlin 默认一个分支处理完就直接跳出多路语句，所以不再需要 break。

（4）关键字 default 被 else 取代。

跟优化后的 if/else 一样，Kotlin 中的 when/else 也允许有返回值，所以上面的多路分支代码可优化为如下代码：

```
btn_when_value.setOnClickListener {
    tv_answer.text = when (count) {
        0 -> "凉风有信的谜底是“讽”"
        1 -> "秋月无边的谜底是“二”"
        else -> "好诗，这真是一首好诗"
    }
    count = (count+1) % 3
}
```

以往 Java 在使用 switch/case 时有个限制，就是 case 后面只能跟常量，不能跟变量，否则编译不通过。现在 Kotlin 去掉了这个限制，进行分支处理时允许引入变量判断，当然引入具体的运算表达式也是可以的。引入变量判断的演示代码如下：

```
var odd:Int = 0
var even:Int = 1
btn_when_variable.setOnClickListener {
    tv_answer.text = when (count) {
        odd -> "凉风有信的谜底是“讽”"
        even -> "秋月无边的谜底是“二”"
        else -> "好诗，这真是一首好诗"
    }
    count = (count+1) % 3
}
```

引入变量判断只是 Kotlin 牛刀小试，真正的功能扩充还在后面。原来的 switch/case 机制中，每个 case 仅仅对应一个常量值，如果 5 个常量值都要进入某个分支，就只能并列写 5 个 case 语句，然后才跟上具体的分支处理语句。现在 when/else 机制中便无须如此麻烦了，这 5 个常量值并排在一起用逗号隔开即可，如果几个常量值刚好是连续数字，可以使用“in 开始值..结束值”指定区间范围；举一反三，若要求不在某个区间范围，则使用语句“!in 开始值..结束值”。扩展功能后的多路分支代码举例如下：

```
btn_when_region.setOnClickListener {
    tv_answer.text = when (count) {
        1,3,5,7,9 -> "凉风有信的谜底是“讽”"
        in 13..19 -> "秋月无边的谜底是“二”"
        !in 6..10 -> "当里的当，少侠你来猜猜"
        else -> "好诗，这真是一首好诗"
    }
    count = (count+1) % 20
}
```

上述代码运行后的演示结果如图 3-3～图 3-6 所示，其中图 3-3 所示为第一个分支的界面，图 3-4 所示为第二个分支的界面，图 3-5 所示为第三个分支的界面，图 3-6 所示为最后一个分支（即 else 分支）的界面。

grammar

谜题： 凉风有信，秋月无边。打二字

谜底： 凉风有信的谜底是“讽”

图 3-3 进入第一个点击分支的界面



图 3-4 进入第二个点击分支的界面

grammar

谜题： 凉风有信，秋月无边。打二字

谜底： 好诗，这真是一首好诗

图 3-5 进入第三个点击分支的界面

grammar

谜题： 凉风有信，秋月无边。打二字

谜底： 当里的当，少侠你来猜猜

图 3-6 进入第四个点击分支的界面

3.1.3 类型判断

条件分支的精彩还在继续，Kotlin 设定了 when/else 语句不仅仅判断变量值，也可以判断变量的类型，如同 Java 的关键字 instanceof 那样。比如 Java 代码若想知晓某个变量是否为字符串类型，则使用以下代码格式进行判断：

```
if (str instanceof String) {
    ...
}
```

那么在 Kotlin 中，关键字 instanceof 被 is 所取代，下面是类型判断的 Kotlin 代码格式：

```
    if (str is String) {
        ...
    }
```

同时，多路分支的 when/else 语句也支持类型判断，只不过在分支判断时采取“is 变量类型->”这种形式。下面是演示类型判断的 Kotlin 代码，在变量 countType 为 Long、Double、Float 三种类型时做多路判断处理：

```
var countType:Number;
btn_when_instance.setOnClickListener {
    count = (count+1) % 3
    countType = when (count) {
        0 -> count.toLong();
        1 -> count.toDouble()
        else -> count.toFloat()
    }
    tv_answer.text = when (countType) {
        is Long -> "此恨绵绵无绝期"
        is Double -> "树上的鸟儿成双对"
        else -> "门泊东吴万里船"
    }
}
```

上面类型判断代码的演示效果如图 3-7～图 3-9 所示，其中图 3-7 展示类型为 Long 的界面，图 3-8 展示类型为 Double 的界面，图 3-9 展示其他类型的界面。

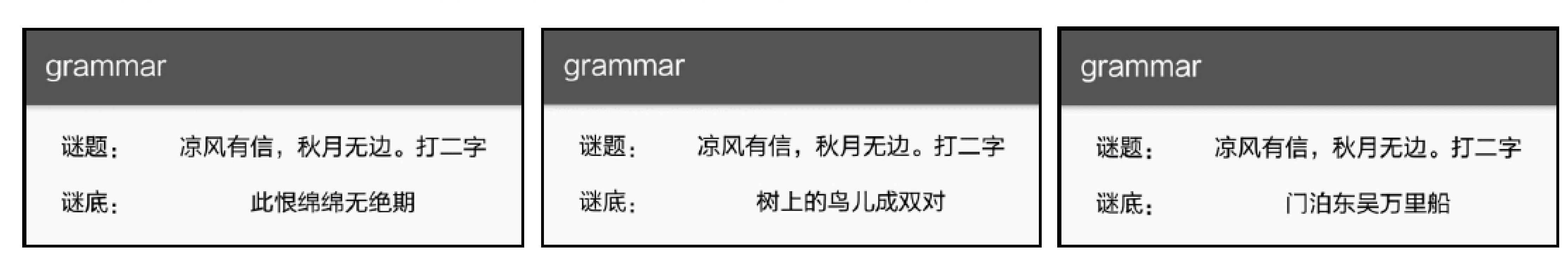


图 3-7　类型为 Long 的分支界面　　图 3-8　类型为 Double 的分支界面　　图 3-9　其他类型的分支界面

总结一下，对于条件分支的处理，Kotlin 实现了简单分支和多路分支，其中简单分支跟 Java 一样都是 if/else 语句，多路分支则由 Java 的 switch/case 语句升级为 when/else 语句。同时，Kotlin 的条件分支允许有返回值，可算是一大改进。另外，Java 的三元运算符“变量名=条件语句?取值 A:取值 B”在 Kotlin 中取消了，对应功能改为使用 if/else 实现；Java 的关键字 instanceof 也取消了，取而代之的是关键字 is，并且多路分支也允许判断变量类型。

3.2　循环处理

3.1 节介绍了简单分支与多路分支的实现，控制语句除了这两种条件分支之外，还有对循环处理的控制，本节接下来继续阐述 Kotlin 如何对循环语句进行操作，看看 Kotlin 引入了哪些新思维。

3.2.1 遍历循环

Koltin 处理循环语句时依旧采纳了 for 和 while 关键字，只是在具体用法上有所微调。首先来看 for 循环，Java 遍历某个队列，可以通过“for (item : list)”形式的语句进行循环操作。同样，Kotlin 也能使用类似形式的循环，区别在于把冒号“:”换成了关键字“in”，具体语句形如“for (item in list)”。下面是 Kotlin 对数组进行循环处理的代码例子：

```
    val poemArray:Array<String> = arrayOf("朝辞白帝彩云间", "千里江陵一日还", "两岸猿
声啼不住", "轻舟已过万重山")
    btn_repeat_item.setOnClickListener {
        var poem:String=""
        for (item in poemArray) {
            poem = "$poem$item, \n"
        }
        tv_poem_content.text = poem
    }
```

上述代码的目的是将一个诗句数组用逗号与换行符拼接起来，以便在界面上展示完整的诗歌内容。拼接后的诗歌显示界面如图 3-10 所示。

注意到图 3-10 中每行诗句都以逗号结尾，这里有个句号问题，因为每首绝句的第一、三行末尾才是逗号，第二、四行的末尾应该是句号，所以这个循环代码得加以改进，补充对数组下标的判断，如果当前是奇数行，末尾就加逗号；如果当前是偶数行，末尾就加句号。倘若使用 Java 编码，要是涉及下标的循环，基本采取“for (初始的赋值语句; 满足循环的条件判断; 每次循环之后的增减语句)”这般形式，具体实现可参考以下的示例代码：

```
    for (int i=0; i<array.length; i++) {
        ...
    }
```

出人意料的是，Kotlin 废除了“for (初始; 条件; 增减)”这个规则，若想实现上述功能，取而代之的是“for (i in 数组变量.indices)”语句，其中 indices 表示该数组变量的下标数组，每次循环都从下标数组依次取出当前元素的下标。根据该规则判断下标的数值，再分别在句尾添加逗号或者句号，据此改造后的 Kotlin 代码如下所示：

```
    btn_repeat_subscript.setOnClickListener {
        var poem:String=""
        //indices 表示数组变量对应的下标数组
        for (i in poemArray.indices) {
            if (i%2 == 0) {
                poem = "$poem${poemArray[i]}, \n"
            } else {
                poem = "$poem${poemArray[i]}。\n"
            }
        }
```

```
    tv_poem_content.text = poem
}
```

代码修正完毕，重新运行测试应用，正确补充标点的诗歌显示界面如图 3-11 所示。

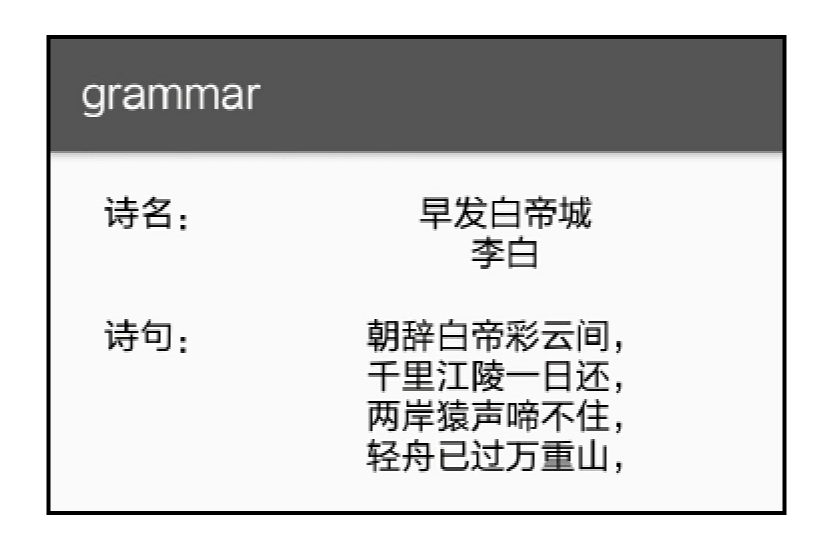


图 3-10　使用逗号简单拼接后的诗歌界面

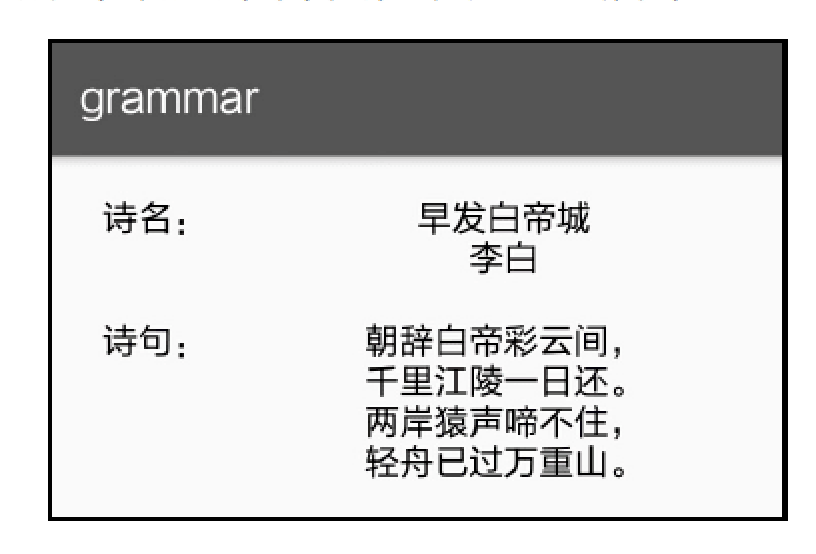


图 3-11　根据下标补充标点的诗歌界面

3.2.2　条件循环

然而 3.2.1 小节取消“for (初始; 条件; 增减)”这个规则是有代价的，因为实际开发中往往存在非同一般的需求，比如对于以下几种情况，Kotlin 的“for (i in 数组变量.indices)”语句就无法很好地处理：

（1）如何设定条件判断的起始值和终止值？

（2）每次循环之后的递增值不是 1 的时候怎么办？

（3）循环方向不是递增而是递减，又如何是好？

（4）与条件判断有关的变量不止一个，怎么办？

（5）循环过程中的变量，在循环结束后还能不能使用？

针对以上情况，其实 Kotlin 也给出了几个解决办法，代价是多了几个诸如 until、step、downTo 这样的关键字，具体的用法例子参见下列代码：

```
// 左闭右开区间，合法值包括 11，但不包括 66
for (i in 11 until 66) { ... }
// 每次默认递增 1，这里改为每次递增 4
for (i in 23..89 step 4) { ... }
// for 循环默认递增，这里使用 downTo 表示递减
for (i in 50 downTo 7) { ... }
```

可是这些解决办法并不完美，因为业务需求是千变万化的，并非限定在几种固定模式。同时，以上规则容易使人混淆，一旦没搞清楚 until 和 downTo 的开闭区间，在判断边界值时就会产生问题。所以更灵活的解决方案是，起止数值、条件判断、循环方向与递增值都应当在代码中明确指定，“for (初始; 条件; 增减)”这个规则固然废除了，但是开发者依旧能够使用 while 语句实现相关功能，所幸 Kotlin 的 while 循环与 Java 的处理是一致的，下面是使用 while 进行循环判断的代码例子：

```
btn_repeat_begin.setOnClickListener {
    var poem:String=""
    var i:Int = 0
```

```
        while (i < poemArray.size) {
            if (i%2 ==0) {
                poem = "$poem${poemArray[i]}，\n"
            } else {
                poem = "$poem${poemArray[i]}。\n"
            }
            i++
        }
        poem = "${poem}该诗歌一共有${i}句。"
        tv_poem_content.text = poem
    }
```

既然 while 语句保留了下来，do/while 语句继续保留，写法跟 Java 相比也没什么变化，采用 do/while 写法的代码如下所示：

```
    btn_repeat_end.setOnClickListener {
        var poem:String=""
        var i:Int = 0
        do {
            if (i%2 ==0) {
                poem = "$poem${poemArray[i]}，\n"
            } else {
                poem = "$poem${poemArray[i]}。\n"
            }
            i++
        } while (i < poemArray.size)
        poem = "${poem}该诗歌一共有${i}句。"
        tv_poem_content.text = poem
    }
```

3.2.3 跳出多重循环

前面的循环处理其实都还中规中矩，只有内忧没有外患，但要是数组里的诗句本身就不完善，比如有空指针、空串、空格串、多余串等，此时就得进行诗句的合法性判断，如此方可输出正常的诗歌文字。前述诗歌例子的合法性判断主要由以下两块代码组成：

（1）如果发现有空指针、空串、空格串，就忽略此行，即使用关键字 continue 继续下一个循环。

（2）如果合法诗句达到 4 句，那么无论是否遍历完成，直接拼好绝句并结束循环，即使用关键字 break 跳出循环。

加入合法性判断的 Kotlin 代码如下，其中主要演示 continue 和 break 的用法：

```
    val poem2Array:Array<String?> = arrayOf("朝辞白帝彩云间", null, "千里江陵一日还",
"", "两岸猿声啼不住", "   ", "轻舟已过万重山", "送孟浩然之广陵")
    btn_repeat_continue.setOnClickListener {
```

```
        var poem:String=""
        var pos:Int=-1
        var count:Int=0
        while (pos <= poem2Array.size) {
            pos++
            //若发现该行是空串或者空格串，则忽略该行
            if (poem2Array[pos].isNullOrBlank())
                continue
            if (count%2 ==0) {
                poem = "$poem${poem2Array[pos]}，\n"
            } else {
                poem = "$poem${poem2Array[pos]}。\n"
            }
            count++
            //若合法行数达到 4 行，则结束循环
            if (count == 4)
                break
        }
        tv_poem_content.text = poem
    }
```

看来合法性判断用到的 continue 和 break，Kotlin 并没有做什么改进呀？这是真的吗？如果是真的，那真是“图样图森破”。以往使用 Java 操作多层循环的时候，有时在内层循环发现某种状况，就得跳出包括外层循环在内的整个循环。例如遍历诗歌数组，一旦在某个诗句中找到“一”字，便迅速告知外界“我中奖啦”之类的欢呼。可是这里有两层循环，如果使用 Java 编码，只能先跳出内层循环，然后外层循环通过判断标志位再决定是否跳出，而不能从内层循环直接跳出外层循环。

现在 Kotlin 大笔一挥，不用这么麻烦，咱想跳到哪里就跳到哪里，只消给外层循环加个@标记，接着遇到情况便直接跳出到这个标记，犹如孙悟空蹦上筋斗云，想去哪儿就去哪儿，多方便。这个创意真好，省事省力省心，赶紧看看下面的 Kotlin 代码是怎么实现的：

```
        btn_repeat_break.setOnClickListener {
            var i:Int = 0
            var is_found = false
            //给外层循环加个名叫 outside 的标记
            outside@ while (i < poemArray.size) {
                var j:Int = 0
                var item = poemArray[i];
                while ( j < item.length) {
                    if (item[j] == '一') {
                        is_found = true
                        //发现情况，直接跳出 outside 循环
                        break@outside
                    }
                    j++
                }
```

```
    //              //如果内层循环直接跳出两层循环，那么下面的判断语句就不需要了
    //              if (is_found)
    //                  break
                i++
            }
            tv_poem_content.text = if (is_found) "我找到'一'字啦" else "没有找到'一'
字呀"
        }
```

总结一下，对于循环语句的操作，Kotlin 仍然保留 for 和 while 两种循环，主要区别在于：Kotlin 取消了“for (初始; 条件; 增减)”这个规则，同时新增了对跳出多重循环的支持（通过“break@标记位”实现）。

3.3 空 安 全

3.2.3 小节末尾介绍多重循环的跳出操作时，演示了发现空串则直接继续下一循环，当时初始化字符串数组使用了表达式“val poem2Array:Array<String?> = ***”，该表达式不免令人疑惑，为何这里要在String后面加个问号？由此，本节就Kotlin如何判断和处理空值再做进一步的深入探讨。

3.3.1 字符串的有效性判断

以往的开发工作中少不了要跟各种异常做斗争，常见的异常种类包括空指针异常 NullPointerException、数组越界异常 IndexOutOfBoundsException、类型转换异常 ClassCastException 等。其中，最让人头痛的当数空指针异常，该异常频繁发生却又隐藏很深。一旦调用某个空对象的方法，就会产生空指针异常。可是 Java 编码的时候编译器不会报错，开发者通常也意识不到问题，只有 App 运行之时发生闪退，查看崩溃日志才会恍然大悟，“原来这里得加上变量非空的判断”。

问题的症结在于，Java 编译器不会检查空值，只能由开发者在代码中手工增加“if (*** != null)”的分支判断。但是业务代码里面的方法调用浩若繁星，倘若在每个方法调用之前都加上非空判断，势必大量代码都充满了“if (*** != null)”。这样做的后果不仅降低了代码的可读性，而且给开发者带来不少的额外工作量。

此外，空指针只是狭义上的空值，广义上的空值除了空指针外，还包括其他开发者认可的情况。比如说 String 类型，字符串的长度为 0 时也可算是空值；如果字符串的内容全部由空格组成，某种意义上也是空值。那么对于字符串的非空判断，用 Java 书写见下面的示例代码：

```
        if (str!=null && str.length()>0 && str.trim().length()>0) {
            ......
        }
```

可以看到，以上的非空判断语句有点冗长了，因此作为开发者，必须把会被多次调用的代码封装成工具类。既然大家都这么想，Android 系统的研发工程师也不例外，所以安卓的 SDK 已经

提供了“TextUtils.isEmpty(***)”这个公共方法，专门用于校验某个字符串是否为空值。Kotlin 的研发人员当然不会放过这点，就像读者在上一节看到的那样，Kotlin 通过 isNullOrBlank 函数对字符串进行空值校验。

下面列出 Kotlin 校验字符串空值的几个方法。

- isNullOrEmpty：为空指针或者字串长度为 0 时返回 true，非空串与可空串均可调用。
- isNullOrBlank：为空指针、字串长度为 0 或者全为空格时返回 true，非空串与可空串均可调用。
- isEmpty：字串长度为 0 时返回 true，只有非空串可调用。
- isBlank：字串长度为 0 或者全为空格时返回 true，只有非空串可调用。
- isNotEmpty：字串长度大于 0 时返回 true，只有非空串可调用。
- isNotBlank：字串长度大于 0 且不是全空格串时返回 true，只有非空串可调用。

3.3.2　声明可空变量

3.3.1 小节的字符串空值校验方法有区分非空串与可空串，这是缘于 Kotlin 引入了空安全的概念，每个类型的变量都分作不可为空和可以为空两种。前面的文章中，正常声明的变量默认都是非空（不可为 null），比如下面声明字符串变量的代码：

```
var strNotNull:String = ""
```

非空变量要么在声明时就赋值，要么在方法调用前赋值；否则未经初始化就调用该变量的方法，Kotlin 会像语法错误那样标红提示：“Variable *** must be initialized”。至于可以为空的变量，可于声明之时在类型后面加个问号，如同“3.2.3 跳出多重循环”声明可空字符串数组的代码“val poem2Array:Array<String?> = ***”。若只声明一个可空字符串变量，则具体的代码例子如下所示：

```
var strCanNull:String?
```

现在定义了两个字符串，其中 strNotNull 为非空串，strCanNull 为可空串。按照前面几个字符串空值校验方法的规则，strNotNull 允许调用全部 6 个方法，但 strCanNull 只允许调用 isNullOrEmpty 和 isNullOrBlank 两个方法。因为变量 strCanNull 可能为空，若去访问一个空字符串的 length 属性，毫无疑问会扔出空指针异常，所以 Kotlin 对可空串增加编译检查，一旦发现某个可空的字符串变量调用了非空方法，比如 isEmpty、isBlank、isNotEmpty、isNotBlank 等，则 Android Studio 立刻标红提示此处存在语法错误：“Only *** calls are allowed on a nullable receiver of type String”。

可是上述的几个 is***方法局限于判断字符串是否为空串，如果要求获得字符串的长度，或者调用其他的字符串方法，此时仍然要判断空指针。以获取字符串长度为例，下面声明三个字符串变量，其中 strA 为非空串，strB 和 strC 都是可空串，不过 strB 为空而 strC 实际有值，字符串变量的声明代码如下：

```
val strA:String = "非空"
val strB:String? = null
val strC:String? = "可空串"
```

对于 strA，因为它是非空串，所以可直接获取 length 长度属性。对于 strB 和 strC 必须进行非空判断，否则编译器会提示该行代码存在错误。这三个字符串的长度获取代码如下所示：

```
    var length:Int = 0
    btn_length_a.setOnClickListener { length=strA.length; tv_check_result.text="
字符串 A 的长度为$length" }
    btn_length_b.setOnClickListener {
        //length=strB.length //这种写法是不行的，因为 strB 可能为空，会扔出空指针异常
        length = if (strB!=null) strB.length else -1
        tv_check_result.text="字符串 B 的长度为$length"
    }
    btn_length_c.setOnClickListener {
        //即使 strC 有值，也必须做非空判断，谁叫它号称可空呢？编译器宁可错杀一千，不可放过一个
        length = if (strC!=null) strC.length else -1
        tv_check_result.text = "字符串 C 的长度为$length"
    }
```

以上代码获取字符串长度的运行界面如图 3-12～图 3-14 所示，其中图 3-12 展示字符串 A 的长度计算结果，图 3-13 展示字符串 B 的长度计算结果，图 3-14 展示字符串 C 的长度计算结果。



图 3-12 字符串 A 的长度计算结果　图 3-13 字符串 B 的长度计算结果　图 3-14 字符串 C 的长度计算结果

3.3.3 校验空值的运算符

虽然使用条件分支可以完成非空判断的功能，可是 Kotlin 仍旧嫌它太啰唆，中国人把繁体字简化为简体字，外国人也想办法简化编程语言，中外人士果然所见略同。既然访问空串的 length 属性会扔出空指针异常，那就加个标记，告诉编译器遇到空指针别扔异常，直接返回空指针就好了，至少避免了处理异常的麻烦。对应的 Kotlin 标记代码如下所示：

```
    var length_null:Int?
    btn_question_dot.setOnClickListener {
        //?.表示变量为空时直接返回 null，所以返回值的变量必须被声明为可空类型
        length_null = strB?.length
        tv_check_result.text = "使用?.得到字符串 B 的长度为$length_null"
    }
```

从以上代码可以看到，这个多出来的标记是个问号，语句“strB?.length”其实等价于“length_null = if (strB!=null) strB.length else null”。但是，该语句意味着返回值仍然可能为空，如果不想在界面上展示“null”，还得另外判断 length_null 是否为空；也就是说，这个做法并未实现与原代码完全一致的功能。

没有完成任务，Kotlin 当然不会罢休，所以它又引入了一个新的运算符“?:”，学名叫作“Elvis 操作符”，叫起来有点拗口，读者可以把它当作是 Java 的三元运算符“变量名=条件语句?取值 A:取值 B”的缩写。引入运算符“?:”的实现代码如下所示：

```
    btn_question_colon.setOnClickListener {
        //?:表示为空时就返回右边的值，即(x!=null)?x.**:y
        length = strB?.length?: -1
        tv_check_result.text = "使用?:得到字符串 B 的长度为$length"
    }
```

这样总该完事了吧？然而执拗的 Kotlin 工程师觉得还是啰嗦，因为经常上一行代码就对字符串 strB 赋值了，所以此时可以百分之百保证 strB 非空，那又何必浪费口舌呢？于是 Kotlin 引入了另一种运算符“!!”，表示甭管那么多，前方没有地雷，弟兄们赶紧上。把双感叹号加在变量名称后面表示强行把该变量从可空类型转为非空类型，从而避免变量是否非空的校验。下面是运算符“!!”的使用代码例子：

```
    btn_exclamation_two.setOnClickListener {
        strB = "排雷完毕"
        length = strB!!.length
        tv_check_result.text = "使用!!得到字符串 B 的长度为$length"
    }
```

既然运算符“!!”强行放弃了非空判断，开发者就得自己注意排雷了。否则的话，一旦出现空指针，App 运行时依然会抛出异常。以下的演示代码在运行时会扔出空指针异常，故而增加了异常捕获处理：

```
    btn_exclamation_two.setOnClickListener {
        //!!表示不做非空判断，强制执行后面的表达式，如果变量为空，就会扔出空异常
        //所以只有在确保为非空时，才能使用!!
        try {
            //即使返回给可空变量 length_null，也会扔出异常
            length = strB!!.length
            tv_check_result.text = "使用!!得到字符串 B 的长度为$length"
        } catch(e:Exception) {
            tv_check_result.text = "发现空指针异常"
        }
    }
```

总结一下，Kotlin 引入了空安全的概念，并在编译时开展变量是否为空的校验。相关的操作符说明概括如下：

（1）声明变量实例时，在类型名称后面加问号，表示该变量可以为空。

（2）调用变量方法时，在变量名称后面加问号，表示一旦变量为空就返回 null。

（3）新引入运算符“?:”，表示一旦变量为空，就返回该运算符右边的表达式。

（4）新引入运算符“!!”，通知编译器不做非空校验。如果运行时发现变量为空，就扔出异常。

3.4 等式判断

等式是编程语言基本的表达式之一，无论哪种高级语言，无一例外都采用双等号“==”判断两个变量是否相等；就算是复杂的变量，在 Java 中也可通过 equals 函数判断两个变量是否相等。按理说这些能够满足绝大多数场合的要求了，那么 Kotlin 又给等式判断加入了哪些新概念呢？下面好好探讨一下各种场合中的等式判断。

3.4.1 结构相等

基本数据类型如整型、长整型、浮点型、双精度、布尔型，无论是在 C/C++还是在 Java 抑或是在 Kotlin，都使用双等号“==”进行两个变量的相等性判断。至于字符串类型则比较特殊，因为最早 C 语言是在内存中开辟一块区域，利用这块区域存储字符串，并返回一个字符指针指向该区域的首地址。此时，如果对两个字符指针进行“==”运算，结果是比较两个指针指向的地址是否相等，而非比较两个地址存储的字符串是否相等。所以 C 语言判断两个字符串是否相等用到了比较字符串专用的 strcmp 函数。

Java 参考了 C++，虽然不再使用字符指针，而使用 String 类型表示字符串，但是 Java 判断两个字符串是否相等依旧采用 equals 函数。从一个函数换成另一个函数，仍然是换汤不换药，没有本质上的改变。

现在 Kotlin 痛定思痛，决心要革除这种沿袭已久的积弊，反正都把字符串当作跟整型一样的基本数据类型，何不直接统一相关的运算操作符呢？因此，既然整型变量之间使用双等号“==”进行等式判断，同理字符串变量之间也能使用双等号“==”来判断。以此类推，判断两个字符串是否不相等通过不等运算符“!=”即可直接辨别。从 Java 到 Kotlin，改变前后的等式判断表达式对照关系见表 3-1。

表 3-1　字符串等值性的 Java 与 Kotlin 判断方式对照关系

字符串的等值性判断要求	Java 的判断方式	Kotlin 的判断方式
判断两个字符串是否相等	strA.equals(strB)	strA == strB
判断两个字符串是否不等	!strA.equals(strB)	strA != strB

接下来，通过代码观察一下等式判断的过程。下面是一个 Kotlin 判断字符串相等性的代码例子：

```
val helloHe:String = "你好"
val helloShe:String = "妳好"
btn_equal_struct.setOnClickListener {
    if (isEqual) {
        tv_check_title.text = "比较 $helloHe 和 $helloShe 是否相等"
        //比较两个字符串是否相等的 Java 写法是 helloHe.equals(helloShe)
        val result = helloHe == helloShe
```

```
            tv_check_result.text = "==的比较结果是$result"
        } else {
            tv_check_title.text = "比较 $helloHe 和 $helloShe 是否不等"
            //比较两个字符串是否不等的 Java 写法是 !helloHe.equals(helloShe)
            val result = helloHe != helloShe
            tv_check_result.text = "!=的比较结果是$result"
        }
        isEqual = !isEqual
    }
```

上述代码的字符串相等判断运行界面如图 3-15 和图 3-16 所示，其中图 3-15 展示 helloHe 与 helloShe 是否相等的判断结果，图 3-16 展示 helloHe 与 helloShe 是否不等的判断结果。

grammar
判断条件： 比较 你好 和 妳好 是否相等
判断结果： ==的比较结果是false

图 3-15 两个字符串是否相等的判断结果

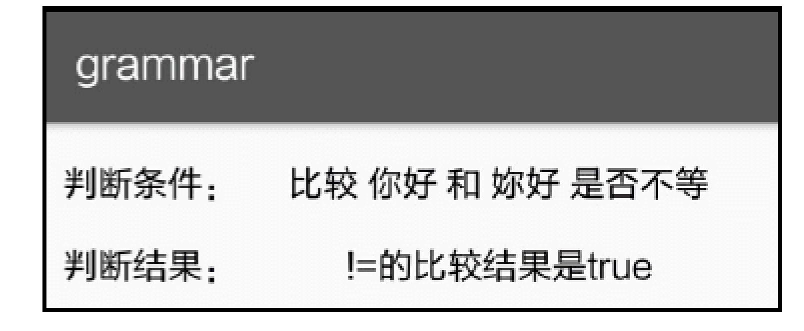


图 3-16 两个字符串是否不等的判断结果

推而广之，不单单字符串 String 类型，凡是 Java 中实现了 equals 函数的类，其变量均可在 Kotlin 中通过运算符“==”和“!=”进行等式判断。这种不比较存储地址，而是比较变量结构内部值的行为，Kotlin 称之为结构相等，即模样相等，通俗地说就是一模一样。

3.4.2 引用相等

有时候仅仅判断两个变量值是否相等并不足以完成某种一致性判断，现实生活中还有更严格的真伪鉴定需求，比如真假美猴王、文物的真品与赝品、兰亭集序的真迹与摹本等。倘若按照结构相等的判断标准，复制品和真品在外观上没有区别，毫无疑问就是相等的。但这个相等的比较结果明显与大众的认知相悖，因为真品是唯一的，复制品再怎么逼真也不可能与真品等价，所以结构相等并不适用于真伪鉴定。判断真伪需要另一种由内而外全部相等的判断准则，该准则叫作引用相等，意思是除了值相等以外，还要求引用的地址（即存储地址）也必须相等。

在 Kotlin 中，结构相等的运算符是双等号“==”，那么引用相等的运算符便是三个等号“===”，多出来的一个等号表示连地址都要相等；结构不等的运算符是“!=”，相对应地，引用不等的运算符是“!==”。不过在大多数场合，结构相等和引用相等的判断结果是一致的。下面列出几种常见的等式判断情景。

（1）对于基本数据类型，包括整型、浮点型、布尔型、字符串，结构相等和引用相等没有区别。

（2）同一个类声明的不同变量，只要有一个属性不相等，则其既是结构不等，也是引用不等。

（3）同一个类声明的不同变量，若 equals 方法校验的每个属性都相等（譬如通过 clone 方法克隆而来的变量复制品），则其结构相等，但引用不等。

为了详细说明以上的等式判断过程，下面给出具体的代码例子，利用系统自带的时间 Date 类演示一下结构相等和引用相等的区别：

```
    val date1:Date = Date()
    val date2:Any = date1.clone() //从 date1 原样克隆一份到 date2
    btn_equal_refer.setOnClickListener {
        when (count++%4) {
            0 -> {
                tv_check_title.text = "比较 date1 和 date2 是否结构相等"
                //结构相等比较的是二者的值
                val result = date1 == date2
                tv_check_result.text = "==的比较结果是$result"
            }
            1 -> {
                tv_check_title.text = "比较 date1 和 date2 是否结构不等"
                //结构不等比较的是二者的值
                val result = date1 != date2
                tv_check_result.text = "!=的比较结果是$result"
            }
            2 -> {
                tv_check_title.text = "比较 date1 和 date2 是否引用相等"
                //引用相等比较的是二者是不是同一个东西，即使克隆地一模一样也不是一个东西
                val result = date1 === date2
                tv_check_result.text = "===的比较结果是$result"
            }
            else -> {
                tv_check_title.text = "比较 date1 和 date2 是否引用不等"
                //引用相等倒过来便是引用不等
                val result = date1 !== date2
                tv_check_result.text = "!==的比较结果是$result"
            }
        }
    }
```

上述代码中的日期变量 date2 从 date1 克隆而来，所以二者的值是完全一样的，区别仅仅是存储的地址不同。接着使用双等号“==”以及“!=”进行结构相等判断，运算结果为相等，具体如图 3-17 和图 3-18 所示，其中图 3-17 展示“==”的判断结果，图 3-18 展示“!=”的判断结果。

图 3-17 结构相等的判断结果

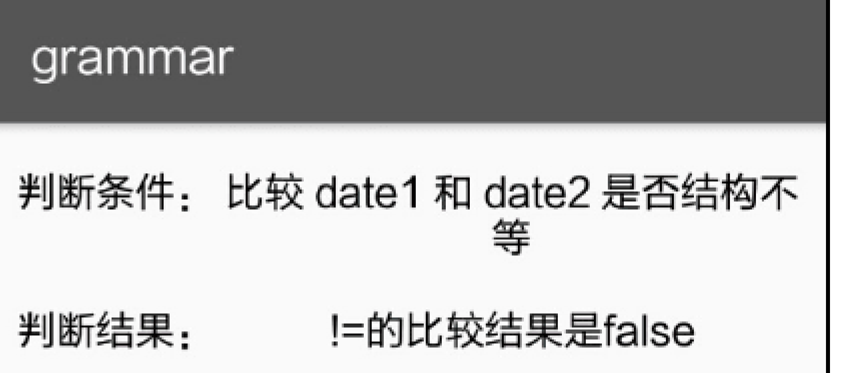


图 3-18 结构不等的判断结果

继续使用三等号“===”以及“!==”进行两个日期变量的引用相等判断，运算结果却是不等，具体如图 3-19 和图 3-20 所示，其中图 3-19 展示“===”的判断结果，图 3-20 展示“!==”的判断结果。

grammar	
判断条件：	比较 date1 和 date2 是否引用相等
判断结果：	===的比较结果是false

图 3-19　引用相等的判断结果

grammar	
判断条件：	比较 date1 和 date2 是否引用不等
判断结果：	!==的比较结果是true

图 3-20　引用不等的判断结果

以上的实验结果证明引用相等校验的是变量的唯一性，而结构相等校验的是变量的等值性。

3.4.3　s 和 in

除了判断两个变量是否相等之外，还有其他维度的等式判断，例如校验变量是否为某种类型、校验数组中是否存在某个元素等，对于这些特殊的等式判断，还得具体问题具体分析。下面对这里举例的两种特殊等式判断进行说明。

1. 运算符 is 和!is

譬如校验变量是否为某种类型，按照 Java 的写法自然采用 instanceof，具体判断语句形如“变量名称 instanceof 类型名称”；如果校验变量是否非某种类型，就需在 instanceof 外层再加上“!”这个非运算符，具体语句形如“!(变量名称 instanceof 类型名称)”。由此可见，Java 的类型判断方式不是太精简，尤其是校验不为某种类型的表达式有点啰唆。

在 Kotlin 中，若要校验变量是否为某种类型，使用的关键字是 is，具体写法形如“变量名称 is 类型名称”；若要校验变量是否不为某种类型，使用的关键字是!is，具体写法形如“变量名称 !is 类型名称”。与 Java 相比，Kotlin 的类型判断足够精炼，表达起来也更加方便。下面来看看 Kotlin 判断变量类型的一个代码例子：

```
val oneLong:Long = 1L
btn_equal_type.setOnClickListener {
    if (isEqual) {
        tv_check_title.text = "比较 oneLong 是否为长整型"
        //is 用于判断是否等于某种类型，对应的 Java 关键字是 instanceof
        val result = oneLong is Long
        tv_check_result.text = "is 的比较结果是$result"
    } else {
        tv_check_title.text = "比较 oneLong 是否非长整型"
        //!is 用于判断是否不等于某种类型
        val result = oneLong !is Long
        tv_check_result.text = "!is 的比较结果是$result"
    }
```

```
        isEqual = !isEqual
    }
```

这个例子校验了变量 oneLong 是否为 Long 长整型数，校验结果分别如图 3-21 和图 3-22 所示，其中图 3-21 展示“is”的判断结果，图 3-22 展示“!is”的判断结果。

grammar

判断条件： 比较 oneLong 是否为长整型

判断结果： is的比较结果是true

图 3-21 “is”的判断结果

grammar

判断条件： 比较 oneLong 是否非长整型

判断结果： !is的比较结果是false

图 3-22 “!is”的判断结果

2. 运算符 in 和!in

还有另一种特殊的等式判断，是校验数组中是否存在某个元素。倘若由 Java 来实现，并没有现成的可用运算符，只能循环遍历该数组，逐个进行数组元素的等式判断，无疑是大费周章。现在有了 Kotlin，则可直接使用关键字 in 来校验，通过“变量名 in 数组名”即可判断数组是否存在等值元素，通过“变量名 !in 数组名”即可判断数组是否不存在等值元素。

照例给出数组判断代码，观察一下 Kotlin 如何实现数组内部的等式判断，具体的判断代码示例如下：

```
val oneArray:IntArray = intArrayOf(1, 2, 3, 4, 5)
val four:Int = 4
val nine:Int = 9
btn_equal_item.setOnClickListener {
    when (count++%4) {
        0 -> {
            tv_check_title.text = "比较 $four 是否存在数组 oneArray 中"
            //in 用于判断变量是否位于数组或容器中，Java 判断数组中是否存在某元素只能采取循环遍历的方式
            val result = four in oneArray
            tv_check_result.text = "in 的比较结果是$result"
        }
        1 -> {
            tv_check_title.text = "比较 $four 是否不在数组 oneArray 中"
            //!in 用于判断变量是否不在数组或容器中
            val result = four !in oneArray
            tv_check_result.text = "!in 的比较结果是$result"
        }
        2 -> {
            tv_check_title.text = "比较 $nine 是否存在数组 oneArray 中"
            //in 用于判断变量是否位于数组或容器中
            val result = nine in oneArray
            tv_check_result.text = "in 的比较结果是$result"
        }
```

```
            else -> {
                tv_check_title.text = "比较 $nine 是否不在数组 oneArray 中"
                //!in 用于判断变量是否不在数组或容器中
                val result = nine !in oneArray
                tv_check_result.text = "!in 的比较结果是$result"
            }
        }
    }
```

以上数组校验代码的实验结果如图 3-23～图 3-26 所示，其中图 3-23 展示数组对 four 变量值进行“in”操作的校验结果，图 3-24 展示数组对 four 变量值进行“!in”操作的校验结果，图 3-25 展示数组对 nine 变量值进行“in”操作的校验结果，图 3-26 展示数组对 nine 变量值进行“!in”操作的校验结果。

grammar
判断条件：比较 4 是否存在数组oneArray中
判断结果：　　in的比较结果是true

图 3-23　对 4 进行“in”操作的校验结果

grammar
判断条件：比较 4 是否不在数组oneArray中
判断结果：　　!in的比较结果是false

图 3-24　对 4 进行“!in”操作的校验结果

grammar
判断条件：比较 9 是否存在数组oneArray中
判断结果：　　in的比较结果是false

图 3-25　对 9 进行“in”操作的校验结果

grammar
判断条件：比较 9 是否不在数组oneArray中
判断结果：　　!in的比较结果是true

图 3-26　对 9 进行“!in”操作的校验结果

3.5　小　　结

本章介绍了 Kotlin 常见的几种控制语句的具体用法，包括条件分支的两种处理方式、循环处理的几个新特性、空值的校验及相关操作符、等式判断的两类判断方式等。

通过本章的学习，读者应能掌握以下技能：

（1）学会运用 Kotlin 的简单分支和多路分支语句。

（2）学会运用 Kotlin 的遍历循环和条件循环语句。

（3）学会声明可空变量，并对可空变量进行各种处理操作。

（4）学会进行 Kotlin 的结构相等和引用相等判断，以及变量类型判断、数组存在等值元素判断。

第 4 章

函数运用

第 3 章介绍了 Kotlin 的各种控制语句，发现 if 和 when 语句允许有返回值，可是在编程世界里面，原本是函数才有返回值。Kotlin 这么一搞，难不成函数没有用武之地了？恰恰相反，Kotlin 不但没有削弱函数的功能，还给函数带来了不少新理念。本章从函数的基本用法、输入参数变化、特殊函数以及如何增强系统函数等几个方面逐次阐述 Kotlin 为函数提供了哪些新功能。

4.1 函数的基本用法

一段相对独立的代码块通过大括号包起来，再给这段代码块起个名字，便形成了函数的雏形。当然一个完整的函数定义还包括输入参数与输出参数两大要素，加上已有的函数名称，从而构成有名有姓、能吃能拉的数据加工单位。下面就对函数的基本定义、输入参数以及输出参数的类型定义做个概要的说明。

4.1.1 与 Java 声明方式的区别

第 3 章介绍控制语句时，在 setOnClickListener 大括号里面写了大段的代码，这不但导致 onCreate 方法变得很臃肿，而且代码的可读性也变差了。对于此种情况，通常的解决办法是把某段代码挪到一个独立的函数中，然后在原位置调用该函数。这样做的好处很多，不仅增强了代码的可读性，还能多次重复调用函数。

那么 Kotlin 对函数的使用跟 Java 相比，有哪些区别呢？先从最常见的 onCreate 方法入手，看看二者都有哪些区别，下面是 Java 编写的 onCreate 函数代码：

```
@Override
public void onCreate(Bundle savedInstanceState) {
```

```
    ...
}
```

而使用 Kotlin 编写的 onCreate 函数代码如下所示：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
}
```

两相对比，可以看到二者主要有以下几点区别：

（1）Java 使用“@Override”表示该函数重载父类的方法，而 Kotlin 使用小写的“override”在同一行表达重载操作。

（2）Java 使用“public”表示该函数是公共方法，而 Kotlin 默认函数就是公开的，所以省略了关键字“public”。

（3）Java 使用“void”表示该函数没有返回参数，而 Kotlin 不存在关键字“void”，若无返回参数，则不用特别说明。

（4）Kotlin 新增了关键字“fun”，表示这里是函数定义，其格式类似于 Java 的关键字“class”，而 Java 不存在关键字“fun”。

（5）Java 声明输入参数的格式为“变量类型 变量名称”，而 Kotlin 声明输入参数的格式为“变量名称: 变量类型”。

（6）Kotlin 引入了空安全机制，如果某个变量允许为空，就需要在变量类型后面加个问号“?”。

其中第 5 点区别的说明参见第 2 章的“2.1.1 基本类型的变量声明”，第 6 点区别的说明参见第 3 章的“3.3.2 声明可空变量”。

由此看来，Kotlin 与 Java 的函数定义之间还有蛮多差异的，其实不但差异不少，而且 Kotlin 增加了不少新功能，有待我们接下来仔细探索。

4.1.2　输入参数的格式

Kotlin 的函数写法与 Java 的传统写法颇为不同，接下来还是从简单的函数声明开始循序渐进地学习。下面是一个简单的函数定义代码，既没有输入参数也没有输出参数：

```
//没有输入参数，也没有输出参数
fun getDinnerEmpty() {
    tv_process.text = "只有空盘子哟"
    tv_result.text = ""
}
```

这个既无入参也无出参的函数看起来比较容易理解。下面再给出一个增加了输入参数的函数定义：

```
//只有输入参数
fun getDinnerInput(egg:Int, leek:Double, water:String, shell:Float) {
    tv_process.text = "食材包括：两个鸡蛋、一把韭菜、几瓢清水"
```

```
        tv_result.text = ""
    }
```

只要学习了第 2 章的基本数据类型的用法，这个存在入参的函数也就易于接受。在上面代码的基础上，增加允许第三个入参为空，则相应的 Kotlin 代码改写如下：

```
    //输入参数存在空值
    fun getDinnerCanNull(egg:Int, leek:Double, water:String?, shell:Float) {
        tv_process.text = if (water!=null) "食材包括：两个鸡蛋、一把韭菜、几瓢清水" else
"没有水没法做汤啦"
        tv_result.text = ""
    }
```

补充的代码修改是在变量类型后面加上问号，表示该参数可以为空。现在有了定义好的函数，若要在 Kotlin 代码中调用它们，那可一点都没变化，原来在 Java 中怎么调用，在 Kotlin 中一样采取“函数名称(参数列表)”的形式进行调用。调用上述三个函数的 Kotlin 代码举例如下：

```
    btn_input_empty.setOnClickListener { getDinnerEmpty() }
    btn_input_param.setOnClickListener { getDinnerInput(2, 1111.1111, "水林淼",
10000f) }
    btn_input_null.setOnClickListener { getDinnerCanNull(2, 1111.1111, null,
10000f) }
```

以上通过三个按钮的点击事件分别调用三个函数，函数调用效果分别如图 4-1～图 4-3 所示，其中图 4-1 展示调用函数 getDinnerEmpty 的结果，图 4-2 展示调用函数 getDinnerInput 的结果，图 4-3 展示调用函数 getDinnerCanNull 的结果。



图 4-1　没有输入参数的界面

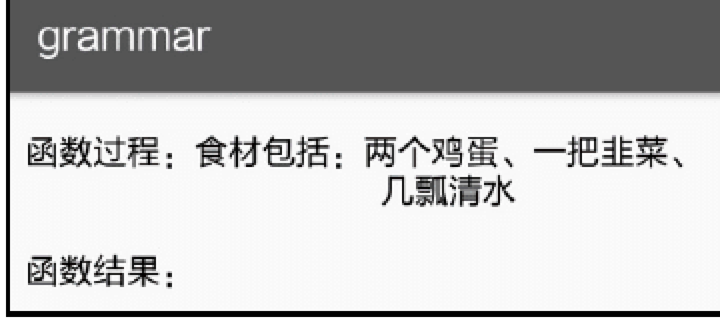


图 4-2　存在输入参数的界面

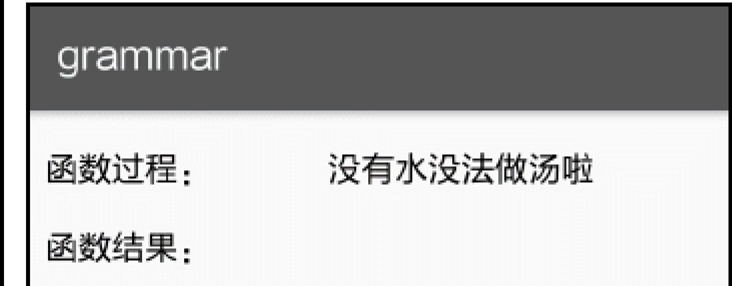


图 4-3　输入参数存在空值的界面

4.1.3　输出参数的格式

4.1.2 小节讨论了存在输入参数的情况，如果函数需要返回输出参数，又该如何是好？在 Java 代码中，函数的返回参数类型在函数名称前面指定，形如“public int main(...)”，但在 Kotlin 中，返回参数类型却在右括号后面指定，形如“fun main(...):Int”。对于习惯了 Java 的开发者而言，Kotlin 的这种写法着实别扭，为了方便记忆，我们姑且把函数当作一种特殊的变量，那么定义函数就跟定义变量是同一种写法。

比如 Kotlin 定义一个整型变量，声明代码如下所示：

```
    var i:Int
```

再看看 Kotlin 如何定义一个函数，具体的函数声明代码如下：

```
   fun main():Int
```

如此一来，功能定义 var 对 fun，参数类型 Int 对 Int，唯一的区别便是函数定义多了一对括号以及括号内部的输入参数。也许这只是巧合，但是偶然中有必然，Kotlin 设计师的初衷正是把函数作为一个特殊的变量，关于这点在本书后面还会再次提到。

既然函数被当作一种特殊的变量，同时每个变量都有变量类型，假如函数存在返回参数，那么自然把返回参数的类型作为该函数的变量类型；要是函数不存在返回参数，也就是 Java 中的返回 void，那该怎么办？这里得澄清一下，Java 使用 void 表示不存在返回参数，然而 Kotlin 的返回参数是一定存在的，即使开发者不声明任何返回参数，Kotlin 函数也会默认返回一个 Unit 类型的对象。

比如前面的函数定义 getDinnerEmpty()，表面上看没有返回任何参数，其实它的真正写法是下面的代码：

```
//Unit 类型表示没有返回参数，也可直接省略 Unit 声明
fun getDinnerUnit():Unit {
    tv_process.text = "只有空盘子哟"
    tv_result.text = ""
}
```

因为 Unit 类型的参数无须开发者返回具体的值，所以 Kotlin 代码往往把函数名称后面的“:Unit”直接省略掉了。增加 Unit 类型的目的是让函数定义完全符合变量定义的形式。若函数需要具体的输出参数，则一样要在函数末尾使用关键字“return”来返回参数值。下面的代码演示如何在函数中返回一个字符串对象：

```
//只有输出参数
fun getDinnerOutput():String {
    tv_process.text = "只有空盘子哟"
    var dinner:String = "巧妇难为无米之炊，汝速去买菜"
    return dinner
}
```

有了以上的各种说明铺垫，现在定义一个同时包含入参和出参的函数，写起代码便顺理成章了。如下所示的代码通过判断各种输入食材，从而输出一道色香味俱全的菜肴：

```
//同时具备输入参数和输出参数
fun getDinnerFull(egg:Int, leek:Double, water:String?, shell:Float):String {
    tv_process.text = if (water!=null) "食材包括：两个鸡蛋、一把韭菜、几瓢清水" else
"没有水没法做汤啦"
    var dinner:String = "两个黄鹂鸣翠柳，\n 一行白鹭上青天。\n 窗含西岭千秋雪，\n 门泊
东吴万里船。"
    return dinner
}
```

存在具体返回参数的函数，调用方式与原来相比并无区别，以下直接给出示例代码：

```
btn_output_empty.setOnClickListener { getDinnerUnit() }
btn_output_param.setOnClickListener { tv_result.text=getDinnerOutput() }
```

```
    btn_full_param.setOnClickListener { tv_result.text=getDinnerFull(2,
1111.1111, "水沝淼", 10000f) }
```

上述三个按钮的点击事件分别调用三个包含返回参数的函数，函数调用效果分别如图 4-4～图 4-6 所示，其中图 4-4 展示调用函数 getDinnerUnit 的结果，图 4-5 展示调用函数 getDinnerOutput 的结果，图 4-6 展示调用函数 getDinnerFull 的结果。

grammar

函数过程： 没有水没法做汤啦

函数结果：

图 4-4　没有输出参数的界面

grammar

函数过程： 只有空盘子哟

函数结果： 巧妇难为无米之炊，汝速去买菜

图 4-5　存在输出参数的界面

grammar

函数过程：食材包括：两个鸡蛋、一把韭菜、几瓢清水

函数结果： 两个黄鹂鸣翠柳，
一行白鹭上青天。
窗含西岭千秋雪，
门泊东吴万里船。

图 4-6　输入参数和输出参数齐全的界面

4.2　输入参数的变化

前面介绍了 Kotlin 对函数的基本用法，包括函数的定义、输入参数的声明、输出参数的声明等，这些足够应付简单的场合了。当然，倘若一门新语言仅仅满足于这些雕虫小技，那也实在没什么前途。既然 Kotlin 志在取代 Java，就必须练成 Java 所不具备的功夫。本节便从函数的输入参数着手，谈谈 Kotlin 对输入参数的改进与增强之处。

4.2.1　默认参数

首先复习一下如何声明函数的输入参数，比如回答“中国的伟大发明有哪些？”这个问题需要定义一个函数，根据输入的几个发明名称将这几个发明拼接成完整的答案。具体的函数定义举例如下：

```
    fun getFourBig(general:String, first:String, second:String, third:String,
fourth:String):String {
        var answer:String = "$general: $first, $second, $third, $fourth"
        return answer
    }
```

该函数的目的是获取中国四大发明的回答，你可以输入中国古代的四大发明，也可以输入外国留学生票选的中国现代四大发明。两种输入对应的函数调用都很简单，只要按照参数顺序依次输入四大发明的名称即可，调用代码如下所示：

```
    var isOdd = true //如果从初始赋值中能够知道变量类型，就无须显式指定该变量的类型
    btn_input_manual.setOnClickListener {
        tv_four_answer.text = if (isOdd) getFourBig("古代的四大发明","造纸术","印刷术",
"火药","指南针") else getFourBig("现代的四大发明","高铁","网购","移动支付","共享单车")
        isOdd = !isOdd
    }
```

以上演示代码的运行结果如图 4-7 和图 4-8 所示，其中图 4-7 所示为奇数次点击时显示古代四大发明的界面，图 4-8 所示为偶数次点击时显示现代四大发明的界面。

可是这么调用函数不够智能，因为中国古代的四大发明人尽皆知，小学生都知道，何必还得每次都手工输入呢？于是 Kotlin 引入了默认参数的概念，允许在定义函数时直接指定输入参数的默认值。如果调用函数时没有给出某参数的具体值，系统就自动对该参数赋予默认值，从而免去每次都要手工赋值的麻烦。默认参数的写法也很简单，只需在声明输入参数时在其后面加上等号及其默认值，详细的函数定义代码如下所示：

```
    fun getFourBigDefault(general:String, first:String="造纸术", second:String="
印刷术", third:String="火药", fourth:String="指南针"):String {
        var answer:String = "$general: $first, $second, $third, $fourth"
        return answer
    }
```

自从有了默认参数，这下函数调用简单多了，就算开发者一时脑袋糊涂想不起来四大发明，也能毫无压力地敲写代码。不信请看下面的调用代码：

```
    btn_input_default.setOnClickListener { tv_four_answer.text=
getFourBigDefault("古代的四大发明") }
```

简化后的代码运行界面如图 4-9 所示，可见默认参数已经派上用场了。

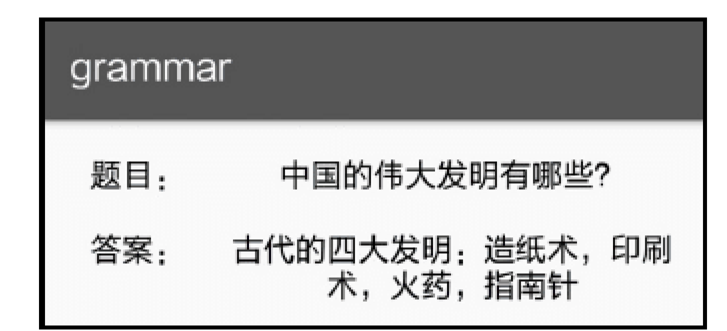


图 4-7　展示古代的四大发明

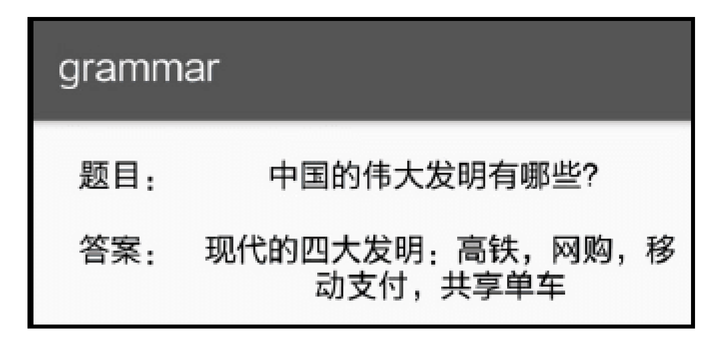


图 4-8　展示现代的四大发明

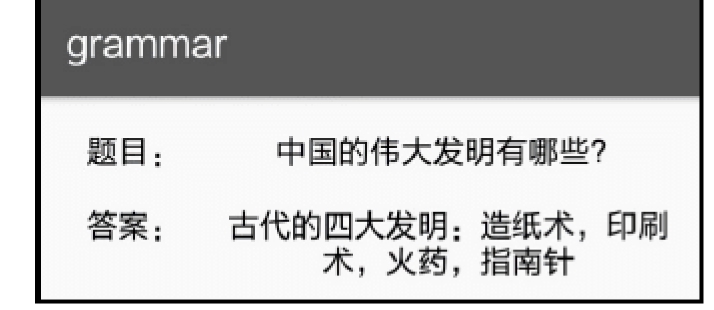


图 4-9　运用默认参数的效果图

4.2.2　命名参数

如果不满意参数的默认值，也可在调用函数时输入新的值，例如四大发明的默认值不包含它们的发明者，现在想增加显示造纸术的发明者蔡伦，则调用 getFourBigDefault 函数时，注意给第二个参数填写指定的描述文字，代码示例如下：

```
    btn_input_part.setOnClickListener { tv_four_answer.text=getFourBigDefault("
古代的四大发明","蔡伦发明的造纸术") }
```

使用新值替换默认值的函数调用效果如图 4-10 所示，可见第一个发明的描述从“造纸术”变成了“蔡伦发明的造纸术”。

有时想要变更的参数并非第一个默认参数，比如第二个默认参数的“印刷术”，虽然印刷术起源于中国是毫无疑义的，但是韩国声称是他们的古人发明了金属活字印刷，德国也有确凿证据证明是古腾堡发明了活字印刷机，这些言论容易误导外人以为中国只是发明了雕版印刷术而已。事实上不光雕版印刷的发明属于中国，就连活字印刷都是北宋的毕昇发明的，所以为了正本清源，“印刷术”的名称可改为影响力更大的“活字印刷”。然而“印刷术”在函数声明里面排在“造纸术”后面，莫非为了给“印刷术”改名，还得把前面的“造纸术”照抄一遍？

为了解决这个不合理的地方，Kotlin 又引进了命名参数的概念，说的是调用函数时可以指定某个参数的名称及其数值，具体格式形如“参数名=参数值”这样。就前述的给“印刷术”改名而言，具体到 Kotlin 编码上面，可参见以下的示例代码：

```
    btn_input_name.setOnClickListener { tv_four_answer.text=getFourBigDefault("
古代的四大发明",second="活字印刷") }
```

由此可见，上面代码使用了命名参数的表达式“second="活字印刷"”，该表达式实现了给指定参数赋值的功能，图 4-11 展示运用命名参数的演示界面。

grammar

题目：　中国的伟大发明有哪些?

答案：　古代的四大发明：蔡伦发明的造纸术，印刷术，火药，指南针

图 4-10　使用新值替换默认值的效果图

grammar

题目：　中国的伟大发明有哪些?

答案：　古代的四大发明：造纸术，活字印刷，火药，指南针

图 4-11　运用命名参数的效果图

4.2.3　可变参数

默认参数结合命名参数的写法至此告一段落。不料吃瓜群众有话说，咱们中国历史悠久、地大物博，伟大发明何止四大发明呢？譬如丝绸、瓷器、茶叶，每个拎出来都是响当当的物件，其地位在古代西方人眼里，好比现代中国人爱买的 LV、劳力士、欧莱雅。所以中国的伟大发明可不能只限于四大发明，必须改成允许随时添加的，想加几个就加几个。

这种随时添加的概念对应于函数定义里面的可变参数，在 Java 体系中，可变参数采用“Object... args”的形式；在 Kotlin 体系中，新增了关键字 vararg，表示其后的参数个数是不确定的。以可变的字符串参数为例，Java 的写法为“String... args”，而 Kotlin 的写法为“vararg args: String?”。函数内部在解析的时候，Kotlin 会把可变参数当作一个数组，开发者需要循环取出每个参数值进行处理，对应的 Kotlin 演示代码如下所示：

```
    fun getFourBigVararg(general:String, first:String="造纸术", second:String="
印刷术", third:String="火药", fourth:String="指南针", vararg otherArray:
String?):String {
        var answer:String = "$general: $first, $second, $third, $fourth"
        //循环取出可变参数包含的所有字段
        for (item in otherArray) {
            answer = "$answer, $item"
        }
```

```
        return answer
    }
```

这下好了，同一个函数既可以输入四大发明，又可以输出七大发明，哪天你给弄个十大发明也是允许的。下面是带有可变参数的函数调用代码：

```
    btn_param_vararg.setOnClickListener {
        //可变参数输入了三个字符串，即"丝绸","瓷器","茶叶"
        tv_four_answer.text = if (isOdd) getFourBigVararg("古代的四大发明") else getFourBigVararg("古代的七大发明","造纸术","印刷术","火药","指南针","丝绸","瓷器","茶叶")
        isOdd = !isOdd
    }
```

引入可变参数的函数调用结果如图 4-12 所示，可见四大发明变成了七大发明。

grammar

题目：　中国的伟大发明有哪些?

答案：　古代的七大发明：造纸术，印刷术，火药，指南针，丝绸，瓷器，茶叶

图 4-12　引入可变参数的效果图

话说中国文化博大精深，除了物质上的发明外，另有不少技艺上的发明，例如国画、中医、武术等，哪个不是国之瑰宝？因此，可变参数也要支持输入这些技巧性的发明，当然为了跟物质性的发明区分开，最好分门别类，把物质性的发明分为一组，技巧性的发明分为一组。

如此一来，可变参数就成了可变的数组参数，同样声明数组参数时也要加上 vararg 前缀，告诉编译器后面的数组个数是变化的。对应的函数声明代码修改如下：

```
    fun getFourBigArray(general:String, first:String="造纸术", second:String="印刷术", third:String="火药", fourth:String="指南针", vararg otherArray: Array<String>):String {
        var answer:String = "$general: $first, $second, $third, $fourth"
        //先遍历每个数组
        for (array in otherArray) {
            //再遍历某个数组中的所有元素
            for (item in array) {
                answer = "$answer, $item"
            }
        }
        return answer
    }
```

对于数组形式的可变参数，进行函数调用时得按照数组参数输入，示例代码如下：

```
    btn_param_array.setOnClickListener {
        //可变参数输入了两个数组变量，每个数组都使用 arrayOf 定义
        tv_four_answer.text = if (isOdd) getFourBigArray("古代的四大发明") else getFourBigArray("古代的 N 大发明","造纸术","印刷术","火药","指南针",arrayOf("丝绸","瓷器","茶叶"),arrayOf("国画","中医","武术"))
        isOdd = !isOdd
    }
```

采取数组变量作为可变参数的函数调用结果如图 4-13 所示，可见根据输入信息一共展示了中国古代的 10 个重要发明。

总结一下，Kotlin 引入了默认参数的概念，并加以扩展允许通过命名参数指定修改某个参数值，而 Java 是不存在默认参数概念的。另外，Kotlin 对 Java 的可变参数功能进行了增强，不但支持普通类型的可变参数，而且支持数组类型的可变参数。

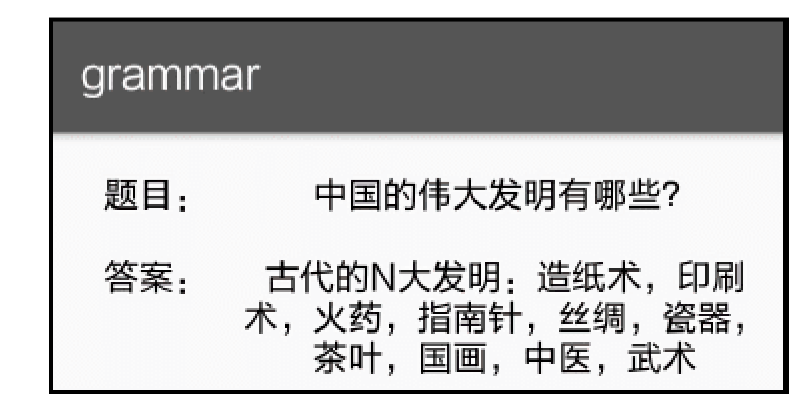


图 4-13　使用数组变量作为可变参数的效果图

4.3　几种特殊函数

4.2 节介绍了 Kotlin 对函数的输入参数所做的增强之处，其实函数这块 Kotlin 还有好些重大改进，集中体现在几类特殊函数，比如泛型函数、内联函数、简化函数、尾递归函数、高阶函数等，因此本节就对这几种特殊函数进行详细说明。

4.3.1　泛型函数

按照之前的例子，函数的输入参数类型必须在定义函数时就要指定，可是有时候参数类型是不确定的，只有在调用函数时方能知晓具体类型，如此一来要怎样声明函数呢？其实在第 2 章的“2.2.1 数组变量的声明”里面就遇到了类似的情况，当时为了采取统一的格式声明基本类型的数组变量，使用“Array<变量类型>”来声明数组变量，并通过 arrayOf 函数获得数组变量的初始值，具体代码如下所示：

```
var int_array:Array<Int> = arrayOf<Int>(1, 2, 3)
var long_array:Array<Long> = arrayOf<Long>(1, 2, 3)
var float_array:Array<Float> = arrayOf<Float>(1.0f, 2.0f, 3.0f)
```

注意到尖括号内部指定了数组元素的类型，这正是泛型的写法“<***>”。由“Array<变量类型>”声明而来的变量可称作泛型变量，至于等号后面的 arrayOf 便是本小节要说的泛型函数。

定义泛型函数时，得在函数名称前面添加“<T>”，表示以 T 声明的参数（包括输入参数和输出参数），其参数类型必须在函数调用时指定。下面举个泛型函数的定义例子，目的是把输入的可变参数逐个拼接起来，并返回拼接后的字符串，示例代码如下：

```
//Kotlin 允许定义全局函数，即函数可在单独的 kt 文件中定义，然后其他地方也能直接调用
fun <T> appendString(tag:String, vararg otherInfo: T?):String {
    var str:String = "$tag: "
    //遍历可变参数中的泛型变量，将其转换为字符串再拼接到一起
    for (item in otherInfo) {
        str = "$str${item.toString()}, "
    }
```

```
        return str
    }
```

调用上面的泛型函数 appendString，就跟调用 arrayOf 方法一样，只需在函数名称后面添加“<变量类型>”即可，然后输入参数照原样填写。以下是 appendString 函数的调用代码例子：

```
        var count = 0
        btn_vararg_generic.setOnClickListener {
            tv_function_result.text = when (count%3) {
                0 -> appendString<String>("古代的四大发明","造纸术","印刷术","火药","指南针")
                1 -> appendString<Int>("小于 10 的素数",2,3,5,7)
                else -> appendString<Double>("烧钱的日子",5.20,6.18,11.11,12.12)
            }
            count++
        }
```

泛型函数 appendString 的演示结果如图 4-14～图 4-16 所示，其中图 4-14 展示输入字符串变量的结果界面，图 4-15 展示输入整型变量的结果界面，图 4-16 展示输入双精度变量的结果界面。

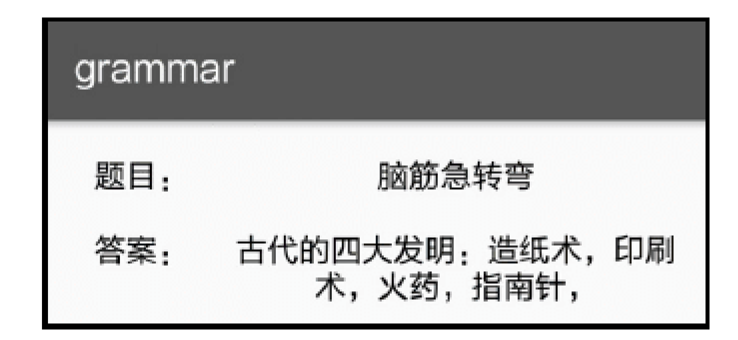


图 4-14　泛型函数输入字符串的效果

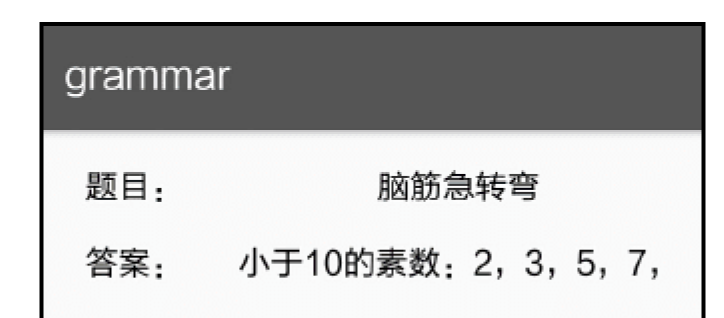


图 4-15　泛型函数输入整型数的效果



图 4-16　泛型函数输入双精度数的效果

4.3.2　内联函数

注意到前面定义泛型函数 appendString 时，是把它作为一个全局函数，也就是在类外面定义，而不在类内部定义。因为类的成员函数依赖于类，只有泛型类（又称模板类）才能拥有成员泛型函数，而普通类是不允许定义泛型函数的，否则编译器会直接报错。不过有个例外情况，如果参数类型都是继承自某种类型，那么允许在定义函数时指定从这个基类泛化开，凡是继承自该基类的子类，都可以作为输入参数进行函数调用，反之则无法调用函数。

举个例子，Int、Float 和 Double 都继承自 Number 类，但是假如定义一个输入参数形式为 setArrayNumber(array:Array<Number>)的函数，它并不接受 Array<Int>或者 Array<Double>的入参。如果要让该方法同时接收整型和双精度的数组入参，就得指定泛型变量 T 来自于基类 Number，即将“<T>”改为“<reified T : Number>”，同时在 fun 前面添加关键字 inline，表示该函数属于内联函数。

内联函数在编译的时候会在调用处把该函数的内部代码直接复制一份，调用 10 次就会复制 10 份，而非普通函数那样仅仅提供一个函数的访问地址。下面是上述例子的内联函数定义代码：

```
        //该函数既不接收 Array<Int>，也不接收 Array<Double>，只好沦为孤家寡人
        fun setArrayNumber(array:Array<Number>) {
```

```
        var str:String = "数组元素依次排列："
        for (item in array) {
            str = str + item.toString() + ", "
        }
        tv_function_result.text = str
    }

    //只有内联函数才可以被具体化
    inline fun <reified T : Number> setArrayStr(array:Array<T>) {
        var str:String = "数组元素依次排列："
        for (item in array) {
            str = str + item.toString() + ", "
        }
        tv_function_result.text = str
    }
```

以上的泛型函数兼内联函数 setArrayStr 在定义的时候稍显麻烦，不过外部的调用方式没有变化，依旧在函数名称后面补充表达式“<变量类型>”。该内联函数的调用代码示例如下：

```
    var int_array:Array<Int> = arrayOf(1, 2, 3)
    var float_array:Array<Float> = arrayOf(1.0f, 2.0f, 3.0f)
    var double_array:Array<Double> = arrayOf(11.11, 22.22, 33.33)
    //Kotlin 进行函数调用时，要求参数类型完全匹配。所以即使 Int 继承自 Number 类，也不能
调用 setArrayNumber 方法传送 Int 类型
    //btn_generic_number.setOnClickListener { setArrayNumber(int_array) }
    btn_generic_number.setOnClickListener {
        when (count%3) {
            0 -> setArrayStr<Int>(int_array)
            1 -> setArrayStr<Float>(float_array)
            else -> setArrayStr<Double>(double_array)
        }
        count++
    }
```

4.3.3 简化函数

在“4.1.3 输出参数的格式”中提到了可将函数当作一种特殊的变量，既然变量通过等号赋值，那么函数也允许使用等号对输出参数赋值。具体地说，如果一个函数的表达式比较简单，一两行代码就可以搞定，那么 Kotlin 允许使用等号代替大括号。

例如，数学上存在计算 n!的阶乘函数，大家都知道 5!=5*4*3*2*1，这个阶乘函数使用 Kotlin 书写的代码格式如下所示：

```
fun factorial(n:Int):Int {
    if (n <= 1) n
```

```
        else n*factorial(n-1)
    }
```

可以看到，阶乘函数类似 Java 中的“判断条件?取值 A:取值 B”三元表达式，只不过内部递归调用函数自身而已。既然 Kotlin 把函数当作一种特殊变量，则允许通过等号给函数这个变量进行赋值，据此可将阶乘函数代码使用等号改写如下：

```
    fun factorial(n:Int):Int = if (n <= 1) n else n*factorial(n-1)
```

4.3.4　尾递归函数

4.3.3 小节的阶乘函数只是一个普通的递归函数，Kotlin 体系还存在一种特殊的递归函数，名叫尾递归函数，它指的是函数末尾的返回值重复调用了自身函数。此时要在 fun 前面加上关键字 tailrec，告诉编译器这是一个尾递归函数，则编译器会相应进行优化，从而提高程序性能。

比如求余弦不动点，即可通过尾递归函数来实现，下面是具体的实现代码例子：

```
    //如果函数尾部递归调用自身，那么可加上关键字 tailrec 表示这是一个尾递归函数
    //此时编译器会自动优化递归，即用循环方式代替递归，从而避免栈溢出的情况
    //比如下面这个求余弦不动点的函数就是尾递归函数
    tailrec fun findFixPoint(x: Double = 1.0): Double
            = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

4.3.5　高阶函数

前面多次提到函数被 Kotlin 当作特殊变量，包括函数声明采取跟变量声明一样的形式“名称:类型”，以及简化函数允许直接用等号连接函数体等，本节最后讲述的内容则是把 A 函数作为 B 函数的输入参数，就像普通变量一样参与 B 函数的表达式计算。此时，因为 B 函数的入参内嵌了 A 函数，故而 B 函数被称作高阶函数，对应的 A 函数则为高阶函数的函数参数，又称函数变量。

为了解释地更加清楚些，接下来看一个例子。对于一个数组变量，若想求得该数组元素的最大值，则可以调用该数组的 max 方法。现在有一个字符串数组，类型为 Array<String>，倘若调用该数组的 max 方法，返回的并非最长的字符串，而是按首字母排序在字母表最靠后的那个字符串。比如有个字符串数组为 arrayOf("How", "do", "you", "do", "I'm　　", "Fine")，调用 max 方法获得的字符串为“you”，而不是长度最长的那个字符串“I'm　　”。

当然，也可以写一个单独的函数专门判断字符串长度，然而要是哪天需要其他比较大小的算法，难道又得再写一个全新的比较函数？显然这么做的代价不菲，所以 Kotlin 引入了高阶函数这个秘密武器，直接把这个比较算法作为参数传进来，由开发者在调用高阶函数时再指定具体的算法函数。就获取字符串数组内部的最大值而言，实现该功能框架的高阶函数代码如下所示：

```
    //允许将函数表达式作为输入参数传进来，就形成了高阶函数，这里的 greater 函数就像是个变量
    //greater 函数有两个输入参数，返回布尔型的输出参数
    //如果第一个参数大于第二个参数，就认为 greater 返回 true，否则返回 false
    fun <T> maxCustom(array: Array<T>, greater: (T, T) -> Boolean): T? {
        var max: T? = null
```

```
        for (item in array)
            if (max == null || greater(item, max))
                max = item
        return max
    }
```

上面高阶函数的第二个参数就是一个函数变量，其中变量名称为 greater，冒号后面的“(T, T)”表示 greater 函数有两个类型为 T 的输入参数，该函数的返回值是 Boolean 类型。现在有了高阶函数的定义，再来看看外部如何调用这个高阶函数，调用的示例代码如下：

```
        var string_array:Array<String> = arrayOf("How", "do", "you", "do", "I'm   ",
"Fine")
        btn_function_higher.setOnClickListener {
            tv_function_result.text = when (count%4) {
                //string_array.max()返回的是 you
                0 -> "字符串数组的默认最大值为${string_array.max()}"
                //因为高阶函数 maxCustom 同时也是泛型函数，所以要在函数名称后面加上<String>
                1 -> "字符串数组按长度比较的最大值为${maxCustom<String>(string_array,
{ a, b -> a.length > b.length })}"
                //string_array.max()对应的高阶函数是 maxCustom(string_array, { a, b ->
a > b })
                2 -> "字符串数组的默认最大值(使用高阶函数)为${maxCustom(string_array, { a,
b -> a > b })}"
                //因为系统可以根据 string_array 判断泛型函数采用了 String 类型，故而函数名称
后面的<String>也可以省略掉
                else -> "字符串数组按去掉空格再比较长度的最大值为${maxCustom
(string_array, { a, b -> a.trim().length > b.trim().length })}"
            }
            count++
        }
```

以上代码在调用 maxCustom 函数时，第二个参数被大括号包了起来，这是 Lambda 表达式的匿名函数写法，中间的“->”把匿名函数分为两部分，前半部分表示函数的输入参数，后半部分表示函数体。“{ a, b -> a.length > b.length }”按照规范的函数写法是下面这样的代码：

```
    fun anonymous(a:String, b:String):Boolean {
        var result:Boolean = a.length > b.length
        return result
    }
```

最后演示一下字符串数组获取最大值的高阶函数调用结果，具体如图 4-17～图 4-20 所示。其中图 4-17 展示字符串数组默认的 max 方法比较结果，此时最大值为“you”；图 4-18 展示依据字符串长度的比较结果，此时最大值为“I'm   ”；图 4-19 展示直接使用大于号比较字符串的结果，此时最大值也为“you”，可见该方式的结果就跟 max 方法一样；图 4-20 展示先将字符串去掉末尾空格，再比较字符串长度的结果，此时最大值为“Fine”。

grammar

题目：　脑筋急转弯

答案：　字符串数组的默认最大值为you

图 4-17　字符串数组的默认比较结果

grammar

题目：　脑筋急转弯

答案：　字符串数组按长度比较的最大值为I'm

图 4-18　高阶函数按长度的比较结果

grammar

题目：　脑筋急转弯

答案：　字符串数组的默认最大值(使用高阶函数)为you

图 4-19　高阶函数按默认的比较结果

grammar

题目：　脑筋急转弯

答案：　字符串数组按去掉空格再比较长度的最大值为Fine

图 4-20　高阶函数修整子串再比较长度的结果

4.4　增强系统函数

前几节主要介绍了 Kotlin 函数的各种概念，并未进行真正的实战演练，假如把 Kotlin 函数应用于实战当中，又有哪些地方需要加以注意呢？本节通过几个具体的实战案例进一步阐述 Kotlin 在函数增强实战中的具体表现。

4.4.1　扩展函数

使用 Java 开发时，虽然系统自带的类已经提供了许多方法，然而经常还是无法完全满足业务需求，此时开发者往往要写一个工具类（比如字符串工具类 StringUtil、日期工具类 DateUtil 等）来补充相关的处理功能，长此以往，工具类越来越多，也越来越难以管理。

基于以上情况，Kotlin 推出了扩展函数的概念，扩展函数允许开发者给系统类补写新的方法，而无须另外编写额外的工具类。比如系统自带的数组 Array 提供了求最大值的 max 方法，也提供了进行排序的 sort 方法，可是并未提供交换数组元素的方法。于是我们打算给 Array 数组类增加新的交换方法，也就是添加一个扩展函数 swap。与普通函数定义不同的是，要在 swap 函数名称前面加上前缀“Array<Int>.”，表示该函数扩展自系统类 Array<Int>。下面是用于交换数组元素的 swap 函数定义代码：

```
fun Array<Int>.swap(pos1: Int, pos2: Int) {
    val tmp = this[pos1] //this 表示数组自身
    this[pos1] = this[pos2]
    this[pos2] = tmp
}
```

不过该函数的缺点是显而易见的，因为它声明了扩展自 Array<Int>，也就意味着只能用于整型数组，不能用于包括浮点数组、双精度数组在内的其他数组。所以，为了增强交换函数的通用性，

必须把 swap 改写为泛型函数，即尖括号内部使用 T 代替 Int。将 swap 方法改写为泛型函数的代码如下：

```
//扩展函数结合泛型函数能够更好地扩展函数功能
fun <T> Array<T>.swap(pos1: Int, pos2: Int) {
    val tmp = this[pos1] //this 表示数组变量自身
    this[pos1] = this[pos2]
    this[pos2] = tmp
}
```

有了扩展函数之后，数组变量可以直接调用新增的 swap 方法，仿佛该函数是系统自带的方法，用起来毫不费劲，真是开发者的福音。以下是 swap 函数调用的代码例子，每调用一次 swap 方法，就把该数组的第一个元素和第四个元素进行交换：

```
    //val array:Array<Int> = arrayOf(1, 2, 3, 4, 5)
    val array:Array<Double> = arrayOf(1.0, 2.0, 3.0, 4.0, 5.0)
    btn_function_extend.setOnClickListener {
        //下标为 0 和 3 的两个数组元素进行交换
        //array 可以是整型数组，也可以是双精度数组
        array.swap(0, 3)
        setArrayStr<Double>(array)
    }
```

swap 函数交换数组元素的运行效果如图 4-21 和图 4-22 所示，其中图 4-21 所示为交换前的界面截图，图 4-22 所为交换后的界面截图，可以看到第一个元素“1.0”与第四个元素“4.0”被交换了过来。

grammar

运算结果： 数组元素依次排列：1.0, 2.0, 3.0, 4.0, 5.0,

图 4-21 数组元素交换前的界面

grammar

运算结果： 数组元素依次排列：4.0, 2.0, 3.0, 1.0, 5.0,

图 4-22 数组元素交换后的界面

4.4.2 扩展高阶函数

“4.3.5 高阶函数”小节中提到的 maxCustom 同时结合了高阶函数和泛型函数的写法，其实还可以给它加上扩展函数的功能。由于该函数的目的是求数组元素的最大值，因此不妨将该函数扩展到 Array<T>中去，扩展后的高阶函数代码示例如下：

```
fun <T> Array<T>.maxCustomize(greater: (T, T) -> Boolean): T? {
    var max: T? = null
    for (item in this)
        if (max == null || greater(item, max))
            max = item
    return max
}
```

相对应地，新的 maxCustomize 将作为数组变量的方法进行调用，而非前面的 maxCustom 那样把数组变量作为入参。下面是改写后的扩展函数调用代码：

```
        btn_function_higher.setOnClickListener {
            tv_function_result.text = when (count%4) {
                0 -> "字符串数组的默认最大值为${string_array.max()}"
                //下面是结合高阶函数与扩展函数的调用代码
                1 -> "字符串数组按长度比较的最大值为${string_array.maxCustomize({ a, b
-> a.length > b.length })}"
                2 -> "字符串数组的默认最大值(使用高阶函数)为
${string_array.maxCustomize({ a, b -> a > b })}"
                else -> "字符串数组按去掉空格再比较长度的最大值为
${string_array.maxCustomize({ a, b -> a.trim().length > b.trim().length })}"
            }
            count++
        }
```

4.4.3　日期时间函数

通过前面两个小节的介绍，使用扩展函数可以很方便地扩充数组 Array 的功能，例如交换两个数组元素、求数组的最大元素等。那么除了数组之外，日期和时间的相关操作也是很常见的，比如获取当前日期、获取当前时间、获取指定格式的日期时间等。因此，基本上每个采取 Java 编码的 Android 工程都需要一个类似 DateUtil.java 的工具类，用于获得不同格式的时间字符串。

下面的 Java 代码便是一个实现日期时间格式化的工具类例子：

```
    public class DateUtil {
        //获取当前完整的日期和时间
        public static String getNowDateTime() {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            return sdf.format(new Date());
        }

        //获取当前时间
        public static String getNowTime() {
            SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
            return sdf.format(new Date());
        }

        //获取当前时间（精确到毫秒）
        public static String getNowTimeDetail() {
            SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss.SSS");
            return sdf.format(new Date());
        }
    }
```

注意到上述代码的时间格式存在大小写字母糅合的情况，为了避免混淆，有必要对这些日期时间格式的定义进行补充说明，具体的时间格式对应关系见表 4-1。

表 4-1 日期时间格式的定义说明

日期时间格式	格式说明
小写的 yyyy	表示 4 位年份数字，如 1949、2017 等
大写的 MM	表示两位月份数字，如 01 表示一月份，12 表示 12 月份
小写的 dd	表示两位日期数字，如 08 表示当月 8 号，26 表示当月 26 号
大写的 HH	表示 24 小时制的两位小时数字，如 19 表示晚上 7 点
小写的 hh	表示 12 小时制的两位小时数字，如 06 可同时表示早上 6 点与傍晚 6 点（因为 12 小时制的表达会引发歧义，所以实际开发中很少这么使用）
小写的 mm	表示两位分钟数字，如 30 表示某点 30 分
小写的 ss	表示两位秒钟数字
大写的 SSS	表示三位毫秒数字

时间格式内部其余的横线“-”、空格“ ”、冒号“:”、点号“.”等字符仅仅是连接符，方便观看各种单位的时间数字而已；在中国，也可采用形如“yyyy 年 MM 月 dd 日 HH 时 mm 分 ss 秒”的时间格式。

现在利用 Kotlin 的扩展函数就无须书写专门的 DateUtil 工具类，直接写几个系统日期类 Date 的扩展函数即可实现日期时间格式转换的功能。改写后的 Date 类扩展函数举例如下：

```
//方法名称前面的 Date.表示该方法扩展自 Date 类
//返回的日期时间格式形如 2017-10-01 10:00:00
fun Date.getNowDateTime(): String {
    val sdf = SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
    return sdf.format(this)
}

//只返回日期字符串
fun Date.getNowDate(): String {
    val sdf = SimpleDateFormat("yyyy-MM-dd")
    return sdf.format(this)
}

//只返回时间字符串
fun Date.getNowTime(): String {
    val sdf = SimpleDateFormat("HH:mm:ss")
    return sdf.format(this)
}

//返回详细的时间字符串，精确到毫秒
fun Date.getNowTimeDetail(): String {
    val sdf = SimpleDateFormat("HH:mm:ss.SSS")
```

```
        return sdf.format(this)
    }

    //返回开发者指定格式的日期时间字符串
    fun Date.getFormatTime(format: String=""): String {
        var ft: String = format
        val sdf = if (!ft.isEmpty()) SimpleDateFormat(ft)
        else SimpleDateFormat("yyyyMMddHHmmss")
        return sdf.format(this)
    }
```

外部调用这些日期类的扩展函数也不会很复杂，以下代码通过“Date().getNowDate()”“Date().getNowTime()”等方法就能获取相应格式的日期和时间字符串：

```
        btn_extend_date.setOnClickListener {
            //以下方法调用自 ExtendDate.kt，采取了扩展函数的方式
            tv_function_result.text = "扩展函数: " + when (count++%5) {
                0 -> "当前日期时间为${Date().getNowDateTime()}"
                1 -> "当前日期为${Date().getNowDate()}"
                2 -> "当前时间为${Date().getNowTime()}"
                3 -> "当前毫秒时间为${Date().getNowTimeDetail()}"
                else -> "当前中文日期时间为${Date().getFormatTime("yyyy 年 MM 月 dd 日 HH
时 mm 分 ss 秒")}"
            }
        }
```

通过扩展函数获取日期时间的效果如图 4-23 和图 4-24 所示，其中图 4-23 展示以标点分隔的日期时间字符串，图 4-24 展示以中文分隔的日期时间字符串。

grammar

运算结果：　扩展函数：当前日期时间为
2017-10-27 00:23:04

图 4-23　以标点分隔的日期时间

grammar

运算结果：　扩展函数：当前中文日期时间为
2017年10月27日00时23分22秒

图 4-24　以中文分隔的日期时间

4.4.4　单例对象

虽然扩展函数已经实现日期信息的获取，但是它的调用方式稍显烦琐，比如“Date().getNowDate()”这个日期方法一共占了 4 个括号，容易使人产生密集恐惧症。况且这些函数必须从某个已存在的类扩展而来，倘若没有可依赖的具体类，也就无法书写扩展函数。所以，Java 编码常见的***Util 工具类，某种程度上反而更灵活、适应面更广，那么 Kotlin 有没有专门的工具类写法呢？

作为一个后起之秀，Kotlin 的设计者显然考虑到了这种情况，并且给出了有针对性的解决方案。在 Java 中，无论是工具类还是实体类抑或是业务类，统统采用 class 关键字，如果是工具类，其内

部的方法都加上 static 修饰符，表示这种方法是静态方法，无须对类进行构造操作即可调用。如此这般，搞得 Java 的 class 像个万金油，什么都能做，却什么都要特殊处理。鉴于此，Kotlin 将工具类的用法提炼了出来，既然这个东西仅仅是作为工具，那么一旦制定了规格就不会再改变了，不能构造也不能修改。故而 Kotlin 使用对象关键字 object 加以修饰，并称之为“单例对象”，其实就相当于 Java 的工具类。

单例对象的用法跟传统的类比较，像是一种阉割了的简化类，倘若把普通类比做 App，则单例对象好比小程序，用完即走，不留下一抹痕迹。譬如前面提到的 getNowDateTime 方法，在单例对象中会分解成两个部分，第一个部分是字符串 nowDateTime 的变量声明，第二个部分是紧跟着的获取变量值的 get 方法。外部访问单例对象的内部变量时，系统会自动调用该变量的 get 方法。

下面是采取单例对象改写后的日期时间工具代码：

```
//关键字 object 用来声明单例对象，就像 Java 中开发者自己定义的 Utils 工具类
//其内部的属性等同于 Java 中的 static 静态属性，外部可直接获取属性值
object DateUtil {

    //声明一个当前日期时间的属性
    //返回的日期时间格式形如 2017-10-01 10:00:00
    val nowDateTime: String
        //外部访问 DateUtil.nowDateTime 时，会自动调用 nowDateTime 附属的 get 方法得到
它的值
        get() {
            val sdf = SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
            return sdf.format(Date())
        }

    //只返回日期字符串
    val nowDate: String
        get() {
            val sdf = SimpleDateFormat("yyyy-MM-dd")
            return sdf.format(Date())
        }

    //只返回时间字符串
    val nowTime: String
        get() {
            val sdf = SimpleDateFormat("HH:mm:ss")
            return sdf.format(Date())
        }

    //返回详细的时间字符串，精确到毫秒
    val nowTimeDetail: String
        get() {
            val sdf = SimpleDateFormat("HH:mm:ss.SSS")
            return sdf.format(Date())
```

```
        }

    //返回开发者指定格式的日期时间字符串
    fun getFormatTime(format: String=""): String {
        val ft: String = format
        val sdf = if (!ft.isEmpty()) SimpleDateFormat(ft)
                  else SimpleDateFormat("yyyyMMddHHmmss")
        return sdf.format(Date())
    }
}
```

外部若要访问单例对象的变量值，直接调用“对象名称.变量名称”即可，此时晃瞎眼的括号都不见踪影，一下子干净了许多。调用单例对象的代码例子如下所示，看起来变得更加简洁了：

```
    btn_object_date.setOnClickListener {
        //以下方法调用自 DateUtil.kt，采取单例对象的方式
        tv_function_result.text = "单例对象: " + when (count++%5) {
            0 -> "当前日期时间为${DateUtil.nowDateTime}"
            1 -> "当前日期为${DateUtil.nowDate}"
            2 -> "当前时间为${DateUtil.nowTime}"
            3 -> "当前毫秒时间为${DateUtil.nowTimeDetail}"
            else -> "当前中文日期时间为${DateUtil.getFormatTime("yyyy 年 MM 月 dd 日
HH 时 mm 分 ss 秒")}"
        }
    }
```

4.5 小　　结

本章介绍了 Kotlin 对函数的几个常见运用方式，包括如何定义一个简单的函数、如何灵活地使用函数的输入参数、几种特殊函数的概念及其用法、如何利用 Kotlin 特性对系统函数进行增强等。

通过本章的学习，读者应能掌握以下技能：

（1）学会定义一个包括输入参数和输出参数在内的完整函数形态。

（2）学会输入参数的几种特殊定义，包括默认参数、命名参数、可变参数，以及如何在外部传送这些特殊的输入参数。

（3）学会常见的几种特殊函数的定义与使用，包括泛型函数、单例函数、简化函数、尾递归函数、高阶函数等。

（4）学会合理利用扩展函数、单例对象等新特性对系统函数进行功能增强。

第 5 章

类和对象

第 4 章末尾提到单例对象可用于实现形态简单的工具类，那么形态各异、血肉丰满的普通类又是怎么实现的呢？为解答这个疑问，本章将好好描述一下 Kotlin 对类和对象的具体用法，并分别从构造、成员、继承、特殊类等几个维度进行详细的分析和介绍。

5.1　类的构造

正所谓高楼大厦平地起，无论是简单的类，还是极其复杂的类，一开头都是从零开始的，再由简单到复杂、由基础到高级。对于类来说，一开始只做一件事：弄块场地搞个开工仪式，表示这事就这么定了，接下来要捋起袖子开干了。类的开工仪式即为“挨踢民工”熟知的构造函数，接下来就阐述类里面构造函数的具体用法。

5.1.1　类的简单定义

先来看看在 Android 开发中多次见过的类 MainActivity，在 Java 代码中该类的写法如下所示：

```
public class MainActivity extends AppCompatActivity {
    ...  //此处省略类的内部代码
}
```

而对应的 Kotlin 代码是下面这样的：

```
class MainActivity : AppCompatActivity() {
    ...  //此处省略类的内部代码
}
```

根据上述代码简单地比较，Kotlin 对类的写法与 Java 之间有以下几点区别：

（1）Kotlin 省略了关键字 public，缘于它默认就是开放的。
（2）Kotlin 用冒号“:”代替 extends，也就是通过冒号表示继承关系。
（3）Kotlin 进行继承时，父类后面多了括号“()”。

表面上二者区别不大，其实类这部分大有玄机，真正用 Kotlin 实现让人出乎意料；接下来要层层剖析，逐步认识 Kotlin 类的真面目。从简单的类定义开始，下面是名为 Animal 的动物类定义的代码：

```
class Animal {
    //类的初始化函数
    init {
        //Kotlin 使用 println 替换 Java 的 System.out.println
        println("Animal：这是个动物的类")
    }
}
```

对应在外部为 Animal 类创建实例的代码如下所示：

```
    btn_class_simple.setOnClickListener {
        //var animal: Animal = Animal()
        //因为根据等号后面的构造函数已经明确知道这是个 Animal 的实例
        //所以声明对象时可以不用指定它的类型
        var animal = Animal()
        tv_class_init.text = "简单类的初始化结果见日志"
    }
```

然后继续给 Kotlin 找茬，不费多少工夫又发现了它跟 Java 的三点不同之处：

（1）Kotlin 对类进行初始化的函数名称叫 init，不像 Java 那样把类名作为构造函数的名称。
（2）Kotlin 打印日志使用类似 C 语言的 println 方法，而非 Java 的 System.out.println。
（3）Kotlin 在创建实例时省略了关键字 new。

其中，初始化函数 init 看似是 Kotlin 对类的构造函数，但它只是构造函数的一部分，并非完整的构造函数。init 方法只定义了初始化操作，却无法直接定义输入参数，因为管理入参定义的另有其人。

5.1.2　类的构造函数

第 4 章介绍函数的时候，提到 Kotlin 把函数看成是一种特殊的变量，那么类在某种意义上算是一种特殊的函数。所以构造函数的输入参数得直接加到类名后面，而 init 方法仅仅表示创建类实例时的初始化动作。下面是添加了入参的类定义代码：

```
//如果主构造函数没有带@符号的注解说明，类名后面的 constructor 就可以省略
//class AnimalMain (context:Context, name:String) {
class AnimalMain constructor(context:Context, name:String) {
    init {
```

```
            context.toast("这是只$name")
        }
    }
```

然而以上代码似乎存在问题，因为一个类可能会有多个构造函数，像自定义视图常常需要定义三个构造函数，例如下面是某个自定义视图的 Java 代码：

```
    public class CustomView extends View {
        public CustomView(Context context) {
            super(context);
        }

        public CustomView(Context context,AttributeSet attrs) {
            super(context, attrs);
        }

        public CustomView(Context context, AttributeSet attrs, int defStyleAttr) {
            super(context, attrs, defStyleAttr);
        }
    }
```

对于上述这种存在多个构造函数的情况，Java 可以通过覆写带不同参数的构造函数来实现，那么 Kotlin 已经在类名后面指明了固定数量的入参，又该如何表示拥有其他参数的构造函数？针对这个疑点，Kotlin 引入了主构造函数与二级构造函数的概念。之前代码演示的只是主构造函数，分为两部分：跟在类名后面的参数是主构造函数的入参，同时 init 方法是主构造函数的内部代码。至于二级构造函数，则可以在类内部直接书写完整的函数表达式。

为了让读者有更直观的认识，下面先贴出一段包含二级构造函数的 Kotlin 类定义代码：

```
    class AnimalMain constructor(context:Context, name:String) {
        init {
            context.toast("这是只$name")
        }

        constructor(context:Context, name:String, sex:Int) : this(context, name) {
            var sexName:String = if(sex==0) "公" else "母"
            context.toast("这只${name}是${sexName}的")
        }
    }
```

从以上代码可以看出，二级构造函数和普通函数相比有两个区别：

（1）二级构造函数没有函数名称，只用关键字 constructor 表示这是一个构造函数。

（2）二级构造函数需要调用主构造函数。“this(context, name)”这句代码在 Java 中要以“super(context, name)”的形式写在函数体内部，在 Kotlin 中则以冒号开头补充到输入参数后面，这意味着二级构造函数实际上是从主构造函数派生而来的，也可看作二级构造函数的返回值是主构造函数。

由此看来，二级构造函数从属于主构造函数，如果使用二级构造函数声明该类的实例，系统就会先调用主构造函数的 init 代码，再调用二级构造函数的自身代码。现在若想声明 AnimalMain 类的实例，既可通过主构造函数声明，也可通过二级构造函数声明，具体的声明代码如下所示：

```
    btn_class_main.setOnClickListener {
        setAnimalInfo()
        when (count%2) {
            0 -> { var animal = AnimalMain(this, animalName) }
            else -> { var animal = AnimalMain(this, animalName, animalSex) }
        }
    }
```

不过在测试过程中发现，通过二级构造函数声明实例有一个问题，就是 toast 会弹窗两次。原因是主构造函数的 init 方法已经弹窗，然后二级构造函数自身再次弹窗，看来这么做并不完美，能否不要强制调用主构造函数呢？为了解决该问题，Kotlin 设定了主构造函数不是必需的，也就是说，类可以把几个构造函数都放在类内部定义，从而都变成二级构造函数，如此就去掉了主构造函数。据此修改之后的类定义代码如下所示：

```
class AnimalSeparate {
    constructor(context:Context, name:String) {
        context.toast("这是只$name")
    }

    constructor(context: Context, name:String, sex:Int) {
        var sexName:String = if(sex==0) "公" else "母"
        context.toast("这只${name}是${sexName}的")
    }
}
```

这样一来，新类 AnimalSeparate 便不存在主构造函数了，两个二级构造函数之间没有从属关系，它们各自的函数代码是互相独立的。无论通过哪个构造函数声明类的实例，都只会调用这个构造函数的代码，而不会像之前那样去调用主构造函数的代码了。

5.1.3　带默认参数的构造函数

未料如此折腾一番，隐隐感觉哪里不对劲，猛然发现改来改去，AnimalSeparate 类依旧完整写着两个构造函数，这么做跟 Java 的构造函数写法又有什么区别呢？无非是把类名换成了关键字 constructor，其他地方仍然换汤不换药。Kotlin 的宗旨是化繁为简，没想到结果却返璞归真了，真是令人吓出一身冷汗。莫急莫急，倘若 Kotlin 黔驴技穷，那么它根本没资格挑战 Java，所以肯定是有办法的。读者是否还记得第 4 章介绍函数时说到的默认参数？类的构造函数同样也能添加默认参数。

注意到 AnimalSeparate 类的两个构造函数只是相差一个输入参数，所以完全可以把它们合并成一个带默认参数的主构造函数，新的主构造函数既能输入两个参数，又能输入三个参数。如果利

用带两个入参的主构造函数创建实例，就形同调用了原来的第一个构造函数“constructor(context: Context, name:String)”；如果利用带三个入参的主构造函数创建实例，就形同调用了原来的第二个构造函数“constructor(context: Context, name:String, sex:Int)”。

下面为主构造函数采取默认参数的类定义代码：

```
//类的主构造函数使用了默认参数
class AnimalDefault (context: Context, name:String, sex:Int = 0) {
    init {
        var sexName:String = if(sex==0) "公" else "母"
        context.toast("这只${name}是${sexName}的")
    }
}
```

这下看起来简洁了许多，新类 AnimalDefault 用起来也毫不费事，之前的实例创建代码只要换个类名就好，完全无缝对接。具体的外部调用代码如下所示：

```
    btn_class_default.setOnClickListener {
        setAnimalInfo()
        when (count%2) {
            0 -> { var animal = AnimalDefault(this, animalName) }
            else -> { var animal = AnimalDefault(this, animalName, animalSex) }
        }
    }
```

构造函数使用默认参数，在 Kotlin 代码中完全运行正常，然而一个项目往往是多人协作开发，“码农”甲写了一个 Kotlin 的类 AnimalDefault，“码农”乙没学过 Kotlin 仍旧用 Java 声明该类的实例，声明该类实例的 Java 代码如下所示：

```
        AnimalDefault animal = new AnimalDefault(this, animalName);
```

原本司空见惯的代码，未曾想编译器居然报错，说什么参数不匹配，这可傻眼了，为什么 Kotlin 用得好好的默认参数，到 Java 那边就不行了呢？这是因为 Java 并不会直接支持默认参数，若想让 Java 也能识别构造函数的默认参数，得往该类的构造函数添加注解“@JvmOverloads”，告知编译器这个类是给 Java 重载用的，好比配备了一个同声翻译机，既能听得懂 Kotlin 代码，又能听得懂 Java 代码。

下面是添加了注解说明的 AnimalDefault 类代码：

```
//加上@JvmOverloads 的目的是让 Java 代码也能识别默认参数
//因为添加了注解标记，所以必须补上关键字 constructor
class AnimalDefault @JvmOverloads constructor(context: Context, name:String,
sex:Int = 0) {
    init {
        var sexName:String = if(sex==0) "公" else "母"
        context.toast("这只${name}是${sexName}的")
    }
}
```

改写后的 AnimalDefault 类通过注解增加了 Java 的接入支持，现在 Java 代码也能够像 Kotlin 那样声明该类的实例了。具体的 Java 声明代码如下所示：

```
        @Override
        public void onClick(View v) {
            if (v.getId() == R.id.btn_class_seperate) {
                setAnimalInfo();
                if (count%2 == 0) {
                    //Java 代码不允许直接支持默认参数
                    //若想让 Java 代码识别默认参数，则需给该类的构造函数添加注解@JvmOverloads
                    AnimalDefault animal = new AnimalDefault(this, animalName);
                } else {
                    //Java 代码必须调用参数完整的构造函数
                    AnimalDefault animal = new AnimalDefault(this, animalName,
animalSex);
                }
            }
        }
```

总结一下，Kotlin 给类的构造函数引进了关键字 constructor，并且区分了主构造函数和二级构造函数。主构造函数的入参在类名后面声明，函数体则位于 init 方法中；二级构造函数从属于主构造函数，它不但由主构造函数派生而来，而且必定先调用主构造函数的实现代码。另外，Kotlin 的构造函数也支持默认参数，从而避免了冗余的构造函数定义。

5.2 类的成员

5.1 节介绍了类的简单定义及其构造方式，当时为了方便观察演示结果，在示例代码的构造函数中直接调用 toast 提示方法，但实际开发是不能这么干的。合理的做法是外部访问类的成员属性或者成员方法，从而获得处理之后的返回值，然后外部再根据返回信息判断对应的处理方式。鉴于此，本节就来谈谈 Kotlin 如何声明成员属性和成员方法，以及外部如何访问类的成员。

5.2.1 成员属性

接上一节动物类的例子，每只动物都有名称和性别两个属性，所以必然要在构造函数中输入这两个参数，对应的类代码如下所示：

```
    class WildAnimal (name:String, sex:Int = 0) {
    }
```

这下有了输入参数，还得声明对应的属性字段，用来保存入参的数值。假如按照 Java 的编码思路，Kotlin 给 WildAnimal 类添加两个属性后的代码应该是下面这样的：

```
class WildAnimal (name:String, sex:Int = 0) {
    var name:String //var 表示动物名称可以修改
    val sex:Int //val 表示动物性别不可修改
    init {
        this.name = name
        this.sex = sex
    }
}
```

要是用惯了 Java 语言，可能觉得上面的写法理所当然，没有什么地方不妥。但你是否想过，以上代码至少有两个冗余之处？

（1）属性字段跟构造函数的入参，二者不但名称一样，并且变量类型也是一样的。

（2）初始化函数中给属性字段赋值，为了区别同名的属性与入参，特意给属性字段添加了前缀“this.”。

你一拍脑袋，嘀咕道：“说的也是”。啰唆是啰唆了一些，可大家都这么写，难不成 Kotlin 还有更短的写法？正所谓细微处见差别，这种看似平常的代码，无意中给程序员带来了不少重复劳动。其实此处的代码逻辑很简单，仅仅是把构造函数的输入参数保存到类的属性中，无论输入参数有几个，该类都依样画瓢地声明同样数量的属性字段并加以赋值。

既然属性字段和构造函数的入参存在一一对应关系，那么可以通过某种机制让编译器自动对其命名与赋值。Kotlin 正是遵循了类似的设计思路，且看下面的 Kotlin 代码是怎样实现的：

```
class WildAnimal (var name:String, val sex:Int = 0) {
}
```

看到 Kotlin 的属性声明代码，会不会觉得很不可思议？与本节开头的类代码相比，只有两个改动之处：其一是给名称参数前面增加了关键字“var”，表示同时声明与该参数同名的可变属性并自动赋值；其二是给性别参数前面增加了关键字“val”，表示同时声明与该参数同名的不可变属性并自动赋值。而改动后的代码的运行结果和手工添加属性声明并赋值的代码是一样的。

比如下面的演示代码，只要声明 WildAnimal 类的对象实例，即可直接访问该对象的名称和性别字段：

```
        btn_member_default.setOnClickListener {
            setAnimalInfo()
            var animal = when (count%2) {
                0 -> WildAnimal(animalName)
                else -> WildAnimal(animalName, animalSex)
            }
            tv_class_member.text = "这只${animal.name}是${if (animal.sex==0) "公" else "母"}的"
        }
```

倘若 WildAnimal 类使用 Java 编码实现，按常规还得补充形如“get***”的属性获取方法，以及形如“set***”的属性设置方法，对应的完整 Java 实现代码如下所示：

```
public class WildAnimal {
    private String name;
    private int sex;

    public WildAnimal(String name, int sex) {
        this.name = namme;
        this.sex = sex;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getSex() {
        return this.sex;
    }

    public void setSex(int sex) {
        this.sex = sex;
    }
}
```

不比不知道，比一比才发现原来 Kotlin 大幅精简了代码，包括：

（1）冗余的同名属性声明语句。

（2）冗余的同名属性赋值语句。

（3）冗余的属性获取方法与设置方法。

看到这里，还有什么理由不好好学习 Kotlin 呢？它既为程序员减少了大量的重复劳动，还有效增强了代码的可读性。

如果某个字段并非入参的同名属性，就需在类内部显式声明该属性字段。例如，前面 WildAnimal 类的性别只是一个整型的类型字段，而界面上展示的是性别的中文名称，所以应当给该类补充一个性别名称的属性字段 sexName，这样每次访问 sexName 字段即可获得动物的性别名称。下面是补充了新属性之后的类定义代码：

```
class WildAnimalMember (var name:String, val sex:Int = 0) {
    //非空的成员属性必须在声明时赋值或者在构造函数中赋值
    //否则编译器会报错“Property must be initialized or be abstract”
    var sexName:String
    init {
        sexName = if(sex==0) "公" else "母"
    }
}
```

现在外部的调用代码可以直接访问新增字段 sexName 了，对应的外部调用代码如下所示：

```
    btn_member_custom.setOnClickListener {
        setAnimalInfo()
        var animal = when (count%2) {
            0 -> WildAnimalMember(animalName)
            else -> WildAnimalMember(animalName, animalSex)
        }
        tv_class_member.text = "这只${animal.name}是${animal.sexName}的"
    }
```

5.2.2 成员方法

类的成员除了成员属性还有成员方法，在类内部定义成员方法的过程类似于第 4 章提到的普通函数定义，具体参见“4.1 函数的基本用法”和“4.2 输入参数的变化”，这里不再赘述。下面给出在动物类中定义成员方法的代码例子，主要增加一个获取动物描述信息的成员方法 getDesc：

```
class WildAnimalFunction (var name:String, val sex:Int = 0) {
    var sexName:String
    init {
        sexName = if(sex==0) "公" else "母"
    }

    fun getDesc(tag:String):String {
        return "欢迎来到$tag：这只${name}是${sexName}的。"
    }
}
```

至于外部调用成员方法的过程，一样是先声明该类的实例，然后通过“实例名称.方法名称(输入参数)”的格式进行函数调用，这种形式跟 Java 相比没什么区别。以上面的 WildAnimalFunction 类为例，外部调用成员方法 getDesc 的具体代码如下所示：

```
    btn_member_function.setOnClickListener {
        setAnimalInfo()
        var animal = when (count%2) {
            0 -> WildAnimalFunction(animalName)
            else -> WildAnimalFunction(animalName, animalSex)
        }
        tv_class_member.text = animal.getDesc("动物园")
    }
```

改为通过成员方法获得加工后的返回信息，Activity 代码就无须自行拼接动物信息字符串了，直接把成员方法的返回值拿来使用即可。上述调用成员方法的演示效果如图 5-1 和图 5-2 所示，其中图 5-1 展示只有一个构造入参的动物描述信息，图 5-2 展示有两个构造入参的动物描述信息。

grammar	
类成员信息：	欢迎来到动物园：这只老虎是公的。

图 5-1　只有一个构造入参的动物信息

grammar	
类成员信息：	欢迎来到动物园：这只斑马是母的。

图 5-2　有两个构造入参的动物信息

5.2.3　伴生对象

前面介绍了 Kotlin 对成员属性和成员方法的处理方式，外部无论是访问成员属性还是访问成员方法，都得先声明类的对象，再通过对象访问类的成员。可是 Java 还有静态成员的概念，静态成员使用关键字 static 来修饰，且外部是通过“类名.静态成员名称”的形式访问静态成员（包括静态属性和静态方法）的。

然而 Kotlin 取消了关键字 static，也就无法直接声明静态成员。为了弥补这方面的功能缺陷，Kotlin 引入了伴生对象的概念，可以把它理解为“影子”，伴生对象之于它所在的类仿佛是如影随形。打个比方，类的实例犹如这个类的孩子，一个类可以拥有很多个孩子；而影子只有一个，并且孩子需要繁衍而来，但影子天生就有、无须繁衍。利用伴生对象的技术可间接实现静态成员的功能，在“5.2.1 成员属性”小节有一个从性别类型获得性别名称的例子，反过来也可以从性别名称获得性别类型，要想实现该功能，可在伴生对象中定义一个静态方法 judgeSex 来判断性别类型。

下面给出使用伴生对象扩充类定义的代码例子：

```
class WildAnimalCompanion (var name:String, val sex:Int = 0) {
    var sexName:String
    init {
        sexName = if(sex==0) "公" else "母"
    }

    fun getDesc(tag:String):String {
        return "欢迎来到$tag：这只${name}是${sexName}的。"
    }

    //在类加载时就运行伴生对象的代码块，其作用相当于 Java 里面的 static { ... }代码块
    //关键字 companion 表示伴随，object 表示对象，WildAnimal 表示伴生对象的名称
    companion object WildAnimal{
        fun judgeSex(sexName:String):Int {
            var sex:Int = when (sexName) {
                "公","雄" -> 0
                "母","雌" -> 1
                else -> -1
            }
            return sex
        }
    }
}
```

以上代码的 judgeSex 方法在输入“公”或者“雄”时，将返回 0；输入“母”或者“雌”时，将返回 1。外部若要调用该方法，则既可使用完整的表达式，形如“WildAnimalCompanion.WildAnimal.judgeSex(名称)”，也可使用简化后的表达式，形如“WildAnimalCompanion.judgeSex(名称)”，后一种方式看起来就等同于 Java 的静态方法调用。下面是外部调用 judgeSex 方法的示例代码：

```
        val sexArray:Array<String> = arrayOf("公","母","雄","雌")
        btn_companion_object.setOnClickListener {
            var sexName:String = sexArray[count++%4]
            //伴生对象的 WildAnimal 名称可以省略掉
            //tv_class_member.text = "\"$sexName\"对应的类型是
${WildAnimalCompanion.WildAnimal.judgeSex(sexName)}"
            tv_class_member.text = "\"$sexName\"对应的类型是
${WildAnimalCompanion.judgeSex(sexName)}"
        }
```

以上利用伴生对象间接实现了 Kotlin 的静态方法，演示代码的运行结果如图 5-3 和图 5-4 所示，其中图 5-3 所示为判断“公”对应性别类型的结果界面，图 5-4 所示为判断“雌”对应性别类型的结果界面。

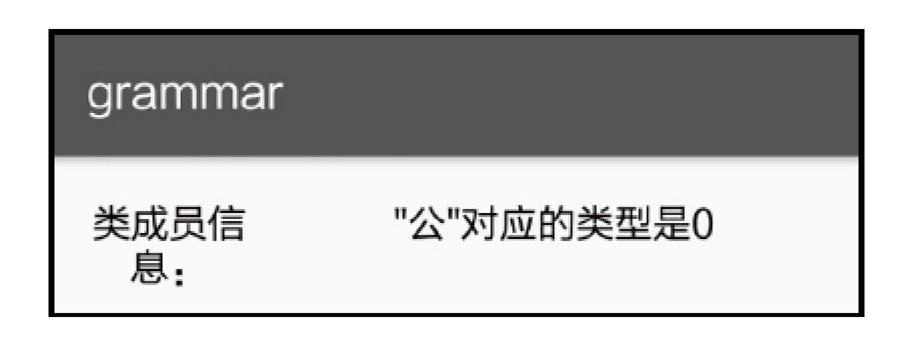


图 5-3　静态方法判断“公”的类型取值

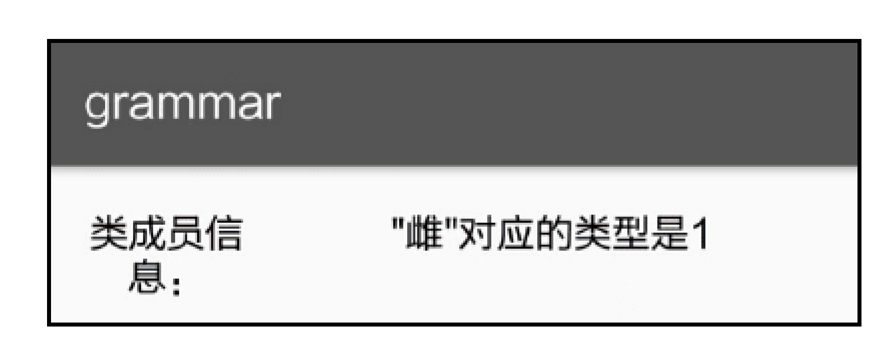


图 5-4　静态方法判断“雌”的类型取值

5.2.4　静态属性

既然伴生对象能够实现静态函数，那么以此类推，同样也能实现静态属性，只要在伴生对象内部增加几个字段定义就行。譬如，judgeSex 方法通过数字 0 表示雄性，通过数字 1 表示雌性，但是只有一个 0 或 1，压根没法联想到是雄性还是雌性，只能凭开发者脑袋的记忆，当然记忆往往会搞混掉。像这种有特定含义的类型数值，更好的办法是采取有实际意义的常量名称，比如在 Android 中存在 Color.RED、Color.GREEN、Color.BLUE 等颜色常量，从它们的名称能够直接联想到颜色含义。

于是动物类表示雄性/雌性的 0 和 1，也可通过静态常量的形式来表达，比如用整型常量 MALE 表示雄性的 0，用 FEMALE 表示雌性的 1。具体到 Kotlin 编码上面，就是在伴生对象中增加这两个常量字段的整型数定义，增加字段定义后的代码示例如下：

```
    class WildAnimalConstant(var name:String, val sex:Int = MALE) {
        var sexName:String
        init {
            sexName = if(sex==MALE) "公" else "母"
        }
```

```
    fun getDesc(tag:String):String {
        return "欢迎来到$tag：这只${name}是${sexName}的。"
    }

    companion object WildAnimal{
        //静态常量的值是不可变的，所以要使用关键字 val 修饰
        val MALE = 0
        val FEMALE = 1
        val UNKNOWN = -1
        fun judgeSex(sexName:String):Int {
            var sex:Int = when (sexName) {
                "公","雄" -> MALE
                "母","雌" -> FEMALE
                else -> UNKNOWN
            }
            return sex
        }
    }
}
```

从以上代码看到，表示性别的数值 0 都被 MALE 代替，数值 1 被 FEMALE 代替，从而提高了代码的可读性。外部若想进行动物性别判断，则可以使用表达式 WildAnimalConstant.MALE 表示雄性，使用 WildAnimalConstant.FEMALE 表示雌性。

总结一下，Kotlin 的类成员分为实例成员与静态成员两种，实例成员包括成员属性和成员方法，其中与入参同名的成员属性可以在构造函数中直接声明，外部必须通过类的实例才能访问类的成员属性和成员方法。类的静态成员包括静态属性与静态方法，它们都在类的伴生对象中定义，外部可以通过类名直接访问该类的静态成员。

5.3　类的继承

5.2 节介绍了类对成员的声明方式与使用过程，从而让读者初步了解了类的成员及其运用。不过在“5.1.1 类的简单定义”中，提到 MainActivity 继承自 AppCompatActivity，而 Kotlin 对于类继承的写法是“class MainActivity : AppCompatActivity() {}”，这跟 Java 对比有明显差异，那么 Kotlin 究竟是如何定义基类并由基类派生出子类呢？为了廓清这些迷雾，本节就对类继承的相关用法进行深入探讨。

5.3.1　开放性修饰符

5.2 节的“5.2.1 成员属性”在演示类成员时多次重写了 WildAnimal 类，这下心急的朋友兴冲冲地准备按照 MainActivity 的继承方式从 WildAnimal 派生出一个子类 Tiger，于是写好构造函数的两个输入参数，并补上基类的完整声明，敲了以下代码，不禁窃喜这么快就大功告成了：

```
class Tiger(name:String="老虎", sex:Int = 0) : WildAnimal(name, sex) {
}
```

谁料编译器无情地蹦出错误提示“The type is final, so it cannot be inherited from”，意思是 WildAnimal 类是 final 类型，所以它不允许被继承。原来 Java 默认每个类都能被继承，除非加了关键字 final 表示终态，才不能被其他类继承。Kotlin 恰恰相反，它默认每个类都不能被继承（相当于 Java 类被 final 修饰了），如果要让某个类成为基类，就需把该类开放出来，也就是添加关键字 open 作为修饰符。

因此，接下来还是按照 Kotlin 的规矩办事，重新写个采取 open 修饰的基类。下面以鸟类 Bird 进行演示，改写后的基类代码框架如下：

```
open class Bird (var name:String, val sex:Int = 0) {
    //此处暂时省略基类内部的成员属性和方法
}
```

现在有了基类框架，还得往里面补充成员属性和成员方法，然后给这些成员添加开放性修饰符。就像读者在 Java 世界中熟知的几个关键字，包括 public、protected、private，分别表示公开、只对子类开放、私有。那么 Kotlin 体系参照 Java 也给出了 4 个开放性修饰符，这些修饰符的取值说明参见表 5-1。

表 5-1 Kotlin 的开放性修饰符的取值说明

开放性修饰符	说明
public	对所有人开放。Kotlin 的类、函数、变量不加开放性修饰符的话，默认就是 public 类型
internal	只对本模块内部开放，这是 Kotlin 新增的关键字。对于 App 开发来说，本模块便是指 App 自身
protected	只对自己和子类开放
private	只对自己开放，即私有

注意到这几个修饰符与 open 一样都加在类和函数前面，并且都包含“开放”的意思，乍看过去还真有点扑朔迷离，到底 open 跟这 4 个开放性修饰符是什么关系？其实很简单，open 不控制某个对象的访问权限，只决定该对象能否繁衍开来，说白了，就是公告这个家伙有没有资格生儿育女。只有头戴 open 帽子的类，才允许作为基类派生出子类来；而头戴 open 帽子的函数，表示它允许在子类中进行重写。如果没戴上 open 帽子，该类就只好打光棍了，无儿无女；函数没戴 open 帽子的话，类的孩子就没法修改它。

至于那 4 个开放性修饰符，则是用来限定允许访问某对象的外部范围，通俗地说，就是哪里的帅哥可以跟这个美女交朋友。头戴 public 的，表示全世界的帅哥都能跟她交朋友；头戴 internal 的，表示只有本国的帅哥可以跟她交朋友；头戴 protected 的，表示只有本单位以及下属单位的帅哥可以跟她交朋友；头戴 private 的，表示肥水不流外人田，只有本单位的帅哥才能跟这个美女交朋友。

因为 private 的限制太严厉了，只对自己开放，甚至都不允许子类染指，所以它跟关键字 open 势同水火。open 表示这个对象可以被继承，或者函数可以被重载，然而 private 却坚决斩断该对象

与其子类的任何关系，因此二者不能并存。倘若在代码中强行给某个方法同时加上 open 和 private，编译器只能无奈地报错“Modifier 'open' is incompatible with 'private'”，意思是 open 与 private 二者不兼容。

5.3.2　普通类继承

按照 5.3.1 小节的开放性相关说明，接下来分别给 Bird 类的类名、函数名、变量名加上修饰符，改写之后的基类代码如下所示：

```
    //Kotlin 的类默认是不能继承的（即 final 类型），如果需要继承某类，该父类就应当声明为 open
类型
    //否则编译器会报错“The type is final, so it cannot be inherited from”
    open class Bird (var name:String, val sex:Int = MALE) {
        //变量、方法、类默认都是 public，所以一般都把 public 省略掉了
        //public var sexName:String
        var sexName:String
        init {
            sexName = getSexName(sex)
        }

        //私有的方法既不能被外部访问，也不能被子类继承，因此 open 与 private 不能共存
        //否则编译器会报错：Modifier 'open' is incompatible with 'private'
        //open private fun getSexName(sex:Int):String {
        open protected fun getSexName(sex:Int):String {
            return if(sex==MALE) "公" else "母"
        }

        fun getDesc(tag:String):String {
            return "欢迎来到$tag：这只${name}是${sexName}的。"
        }

        companion object BirdStatic{
            val MALE = 0
            val FEMALE = 1
            val UNKNOWN = -1
            fun judgeSex(sexName:String):Int {
                var sex:Int = when (sexName) {
                    "公","雄" -> MALE
                    "母","雌" -> FEMALE
                    else -> UNKNOWN
                }
                return sex
            }
        }
    }
```

好不容易鼓捣出来一个正儿八经的鸟儿基类，再来声明一个它的子类试试，例如鸭子是鸟类的一种，于是下面有了鸭子的类定义代码：

```
//注意父类 Bird 已经在构造函数声明了属性，故而子类 Duck 无须重复声明属性
//也就是说，子类的构造函数在输入参数前面不需要再加 val 和 var
class Duck(name:String="鸭子", sex:Int = Bird.MALE) : Bird(name, sex) {
}
```

回到 Activity 页面代码，按以下代码调用新定义的鸭子类试试：

```
    btn_class_duck.setOnClickListener {
        var sexBird = if (count++%3==0) Bird.MALE else Bird.FEMALE
        var duck = Duck(sex=sexBird)
        tv_class_inherit.text = duck.getDesc("鸟语林")
    }
```

鸭子类调用之后的运行效果如图 5-5 和图 5-6 所示，其中图 5-5 展示公鸭信息，图 5-6 展示母鸭信息。

grammar

类继承信息：　欢迎来到鸟语林：这只鸭子是公的。

图 5-5　继承自鸟类的公鸭信息

grammar

类继承信息：　欢迎来到鸟语林：这只鸭子是母的。

图 5-6　继承自鸟类的母鸭信息

子类也可以定义新的成员属性和成员方法，或者重写被声明为 open 的父类方法。比方说性别名称“公”和“母”一般用于家禽，像公鸡、母鸡、公鸭、母鸭等；而指代野生鸟类的性别则通常使用“雄”和“雌”。所以定义野生鸟类的时候，就得重写获取性别名称的 getSexName 方法，把“公”和“母”的返回值改为“雄”和“雌”。

重写 getSexName 方法之后，另外定义一个鸵鸟类的代码如下所示：

```
class Ostrich(name:String="鸵鸟", sex:Int = Bird.MALE) : Bird(name, sex) {
    //继承 protected 的方法，标准写法是“override protected”
    //override protected fun getSexName(sex:Int):String {
    //不过 protected 的方法继承过来默认就是 protected，所以也可直接省略 protected
    //override fun getSexName(sex:Int):String {
    //protected 的方法继承之后允许将可见性升级为 public，但不能降级为 private
    override public fun getSexName(sex:Int):String {
        return if(sex==MALE) "雄" else "雌"
    }
}
```

然后在 Activity 代码中补充鸵鸟类的方法调用，具体的调用代码如下所示：

```
    btn_class_ostrich.setOnClickListener {
        var sexBird = if (count++%3==0) Bird.MALE else Bird.FEMALE
        var ostrich = Ostrich(sex=sexBird)
```

```
        tv_class_inherit.text = ostrich.getDesc("鸟语林")
    }
```

保存代码重新编译运行，可见鸵鸟类的演示结果如图 5-7 和图 5-8 所示，其中图 5-7 展示雄鸵鸟的资料，图 5-8 展示雌鸵鸟的资料。

grammar

类继承信息：　欢迎来到鸟语林：这只鸵鸟是雄的。

图 5-7　继承自鸟类的雄鸵鸟资料

grammar

类继承信息：　欢迎来到鸟语林：这只鸵鸟是雌的。

图 5-8　继承自鸟类的雌鸵鸟资料

5.3.3　抽象类

除了 5.3.2 小节讲的普通类继承，Kotlin 也存在与 Java 类似的抽象类，抽象类之所以存在，是因为其内部拥有被关键字 abstract 修饰的抽象方法。抽象方法没有具体的函数体，故而外部无法直接声明抽象类的实例；只有在子类继承时重写抽象方法，方可使用该子类正常声明对象实例。

举个例子，鸡属于鸟类，可公鸡和母鸡的叫声是不一样的，公鸡是“喔喔喔”地叫，而母鸡是“咯咯咯”地叫。所以鸡这个类的叫唤方法“callOut”发出什么声音并不确定，只能先声明为抽象方法，连带着鸡类“Chicken”也变成抽象类了。

根据鸡类的叫声抽象方案定义一个抽象的 Chicken 类，代码示例如下：

```
    //子类的构造函数，原来的输入参数不用加 var 和 val，新增的输入参数必须加 var 或者 val
    //因为抽象类不能直接使用，所以构造函数不必给默认参数赋值
    abstract class Chicken(name:String, sex:Int, var voice:String) : Bird(name, sex) {
        val numberArray:Array<String> = arrayOf("一","二","三","四","五","六","七","八","九","十");
        //抽象方法必须在子类进行重写，所以可以省略关键字 open，因为 abstract 方法默认就是 open 类型
        //open abstract fun callOut(times:Int):String
        abstract fun callOut(times:Int):String
    }
```

接着从 Chicken 类派生出公鸡类 Cock，指定公鸡的声音为“喔喔喔”，同时还要重写 callOut 方法，明确公鸡的叫唤行为。具体的 Cock 类代码如下所示：

```
    class Cock(name:String="鸡", sex:Int = Bird.MALE, voice:String="喔喔喔") : Chicken(name, sex, voice) {
        override fun callOut(times: Int): String {
            var count = when {
                //when 语句判断大于和小于时，要把完整的判断条件写到每个分支中
                times<=0 -> 0
                times>=10 -> 9
                else -> times
            }
```

```
            return "$sexName$name${voice}叫了${numberArray[count]}声，原来它在报晓呀。"
        }
    }
```

同样派生而来的母鸡类“Hen”也需指定母鸡的声音“咯咯咯”，并重写 callOut 叫唤方法。具体的 Hen 类代码如下所示：

```
    class Hen(name:String="鸡", sex:Int = Bird.FEMALE, voice:String="咯咯咯") :
Chicken(name, sex, voice) {
        override fun callOut(times: Int): String {
            var count = when {
                times<=0 -> 0
                times>=10 -> 9
                else -> times
            }
            return "$sexName$name${voice}叫了${numberArray[count]}声，原来它下蛋了呀。"
        }
    }
```

定义好了 callOut 方法，外部即可调用 Cock 类和 Hen 类的该方法了，调用的例子代码如下所示：

```
        btn_abstract_cock.setOnClickListener {
            //调用公鸡类的叫唤方法
            tv_class_inherit.text = Cock().callOut(count++%10)
        }

        btn_abstract_hen.setOnClickListener {
            //调用母鸡类的叫唤方法
            tv_class_inherit.text = Hen().callOut(count++%10)
        }
```

梳理一下上面的例子，首先定义了一个包含抽象方法 callOut 的抽象类 Chicken；然后由该类派生出两个子类，分别是公鸡类 Cock 和母鸡类 Hen，两个子类都重新实现了自己的 callOut 方法；最后由外部调用重写之后的 callOut 方法。该例子的演示界面如图 5-9 和图 5-10 所示，其中图 5-9 显示公鸡的叫唤行为，图 5-10 显示母鸡的叫唤行为。

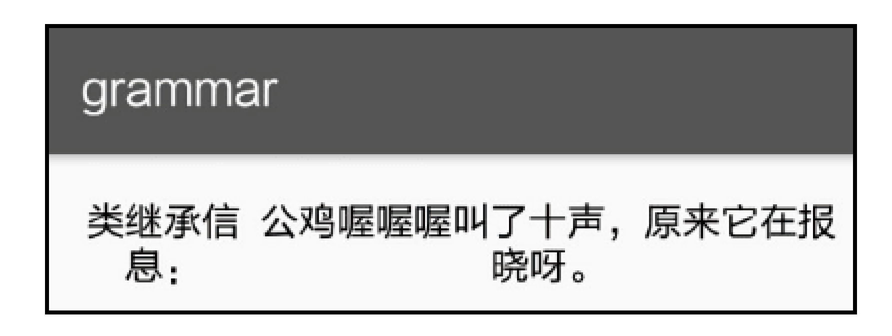


图 5-9　继承自抽象鸡类的公鸡叫声

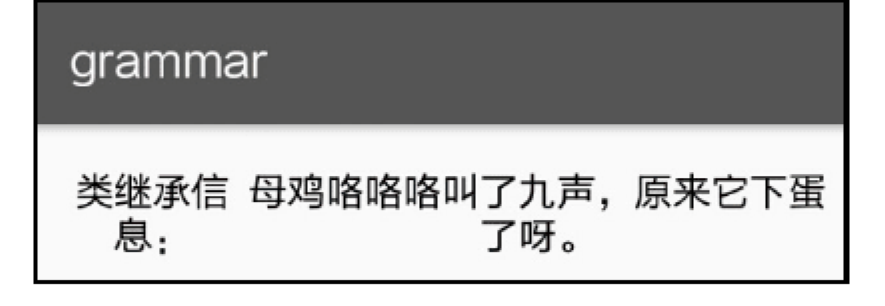


图 5-10　继承自抽象鸡类的母鸡叫声

5.3.4　接口

既然提到了抽象类，就不得不提接口 interface。Kotlin 的接口与 Java 一样是为了间接实现多重继承，由于直接继承多个类可能存在方法冲突等问题，因此 Kotlin 在编译阶段就不允许某个类同

时继承多个基类，否则会报错“Only one class may appear in a supertype list”，意思是继承列表中只允许出现一个类。于是乎，只能通过接口定义几个抽象方法，然后在实现该接口的具体类中重写这几个方法，从而间接实现类似 C++多重继承的功能。

在 Kotlin 中定义接口需要注意以下几点：

（1）接口不能定义构造函数，否则编译器会报错“An interface may not have a constructor”。

（2）接口的内部方法通常要被实现它的类进行重写，所以这些方法默认为抽象类型。

（3）与 Java 不同的是，Kotlin 允许在接口内部实现某个方法，而 Java 接口的所有内部方法都必须是抽象方法。

Android 开发最常见的接口是控件的点击监听器 View.OnClickListener，其内部定义了控件的点击动作 onClick，类似的还有长按监听器 View.OnLongClickListener、选择监听器 CompoundButton.OnCheckedChangeListener 等，它们无一例外都定义了某种行为的事件处理过程。对于本节的鸟类例子而言，也可通过一个接口定义鸟儿的常见动作行为，譬如鸟儿除了叫唤动作外，还有飞翔、游泳、奔跑等动作，有的鸟类擅长飞翔（如大雁、老鹰），有的鸟类擅长游泳（如鸳鸯、鸬鹚），有的鸟类擅长奔跑（如鸵鸟、鸸鹋）。因此针对鸟类的飞翔、游泳、奔跑等动作，即可声明 Behavior 接口，并在该接口中定义几个行为方法，如 fly、swim、run 等。

下面是一个定义好的行为接口的代码例子：

```
    //Kotlin 与 Java 一样不允许多重继承，即不能同时继承两个及两个以上类
    //否则编译器报错“Only one class may appear in a supertype list”
    //所以仍然需要接口 interface 来间接实现多重继承的功能
    //接口不能带构造函数（那样就变成一个类了），否则编译器报错“An interface may not have a constructor”
    //interface Behavior(val action:String) {
    interface Behavior {
        //接口内部的方法默认就是抽象的，所以不加 abstract 也可以，当然 open 也可以不加
        open abstract fun fly():String
        //比如下面这个 swim 方法就没加关键字 abstract，也无须在此处实现方法
        fun swim():String
        //Kotlin 的接口与 Java 的区别在于，Kotlin 接口内部允许实现方法
        //此时该方法不是抽象方法，就不能加上 abstract
        //不过该方法依然是 open 类型，接口内部的所有方法都默认是 open 类型
        fun run():String {
            return "大多数鸟儿跑得并不像样，只有鸵鸟、鸸鹋等少数鸟类才擅长奔跑。"
        }
        //Kotlin 的接口允许声明抽象属性，实现该接口的类必须重载该属性
        //与接口内部方法一样，抽象属性前面的 open 和 abstract 也可省略掉
        //open abstract var skilledSports:String
        var skilledSports:String
    }
```

那么其他类在实现 Behavior 接口时，跟类继承一样把接口名称放在冒号后面，也就是说，Java 的 extends 和 implement 这两个关键字在 Kotlin 中都被冒号取代了。然后就像重写抽象类的抽象方

法一样重写该接口的抽象方法，以定义鹅的 Goose 类为例，重写接口方法之后的代码如下所示：

```
class Goose(name:String="鹅", sex:Int = Bird.MALE) : Bird(name, sex), Behavior {
    override fun fly():String {
        return "鹅能飞一点点，但飞不高，也飞不远。"
    }

    override fun swim():String {
        return "鹅，鹅，鹅，曲项向天歌。白毛浮绿水，红掌拨清波。"
    }

    //因为接口已经实现了 run 方法，所以此处可以不用实现该方法，当然你要实现它也行
    override fun run():String {
        //super 用来调用父类的属性或方法，由于 Kotlin 的接口允许实现方法，因此 super 所指
的对象也可以是 interface
        return super.run()
    }

    //重载了来自接口的抽象属性
    override var skilledSports:String = "游泳"
}
```

这下大功告成，Goose 类声明的鹅不但具备鸟类的基本功能，而且能飞、能游、能跑，活脱脱一只栩栩如生的大白鹅呀。且看下面群鹅千姿百态的调用代码：

```
    btn_interface_behavior.setOnClickListener {
        tv_class_inherit.text = when (count++%3) {
            0 -> Goose().fly()
            1 -> Goose().swim()
            else -> Goose().run()
        }
    }
```

上述群鹅乱舞的界面如图 5-11～图 5-13 所示，其中 5-11 展示鹅展翅扑腾的效果，图 5-12 展示鹅游来游去的效果，图 5-13 展示鹅步履蹒跚的效果。

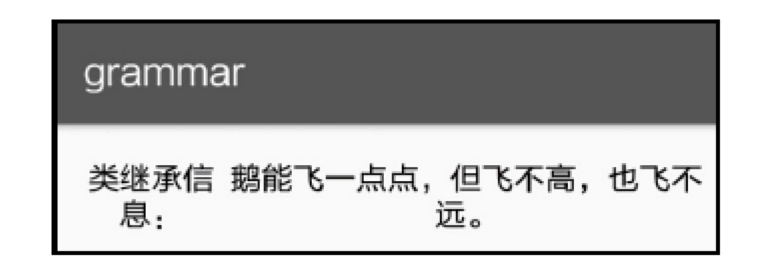


图 5-11　鹅类实现了飞翔接口

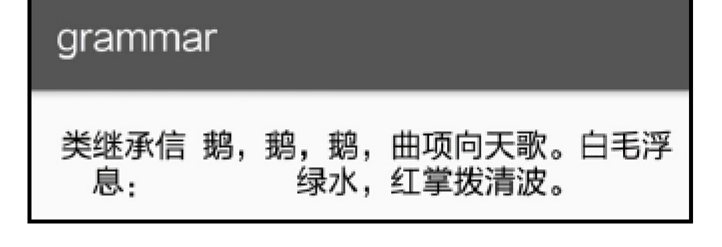


图 5-12　鹅类实现了游泳接口

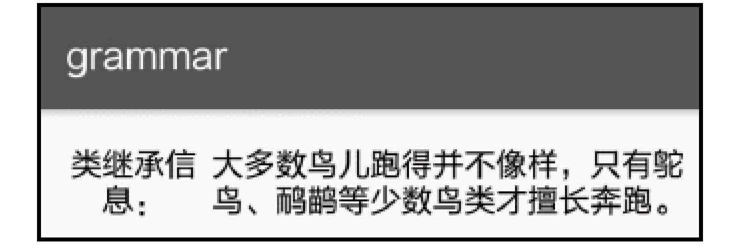


图 5-13　鹅类实现了奔跑接口

5.3.5　接口代理

通过实现接口固然完成了相应行为，但是鸟类这个家族非常庞大，如果每种鸟都要实现 Behavior 接口，可想而知工作量是多么巨大。其实鸟类的行为并不繁多，按照习性区分，主要分为

擅长飞翔的飞禽、擅长游泳的水禽、擅长奔跑的走禽三类。倘若依照通常的处理方式，可从 Bird 类与 Behavior 接口联合派生出三个抽象类，包括飞禽类、水禽类、走禽类，如此貌似解决了重复实现接口的问题。然而处于不同的环境之中，同一只鸟儿可能表现出不同的行为，比如大雁既擅长飞翔又擅长游泳，火鸡既擅长奔跑也能短距离飞翔，这时抽象类的办法就不管用了。

为了让各种鸟类适应上述不同场景的行为要求，Kotlin 引入了接口代理的技术，即一个类先声明继承自某个接口，但并不直接实现该接口的方法，而是把已经实现该接口的代理类作为参数传给前面的类，相当于告诉前面的类："该接口的方法我已经代替你实现了，你直接拿去用便是"。这样做的好处是，输入参数可以按照具体的业务场景传送相应的代理类，也就是说，一只鸟儿怎么飞、怎么游、怎么跑并不是一成不变的，而是由实际情况决定的。譬如，大雁越冬时往南迁移，此时大雁的行为表现是飞禽；迁徙到目的地留下来觅食，此时大雁的行为表现是水禽。

接口代理具体到 Kotlin 编码上，首先要分别定义飞禽、水禽、走禽的三个行为类，下面是飞禽的行为类代码例子：

```
class BehaviorFly : Behavior {
    override fun fly():String {
        return "翱翔天空"
    }

    override fun swim():String {
        return "落水凤凰不如鸡"
    }

    override fun run():String {
        return "能飞干嘛还要走"
    }

    override var skilledSports:String = "飞翔"
}
```

下面是水禽的行为类代码例子：

```
class BehaviorSwim : Behavior {
    override fun fly():String {
        return "看情况，大雁能展翅高飞，企鹅却欲飞还休"
    }

    override fun swim():String {
        return "怡然戏水"
    }

    override fun run():String {
        return "赶鸭子上树"
    }

    override var skilledSports:String = "游泳"
}
```

下面是走禽的行为类代码例子：

```
    class BehaviorRun : Behavior {
        override fun fly():String {
            return "飞不起来"
        }

        override fun swim():String {
            return "望洋兴叹"
        }

        override fun run():String {
            return super.run()
        }

        override var skilledSports:String = "奔跑"
    }
```

接着定义一个引用了代理类的野禽基类，通过关键字 by 表示该接口将由入参中的代理类实现，野禽基类 WildFowl 的定义代码如下所示：

```
    //只有接口才能够使用关键字 by 进行代理操作
    //如果 by 的对象是个类，编译器就会报错“Only interfaces can be delegated to”
    class WildFowl(name:String, sex:Int=MALE, behavior:Behavior) : Bird(name, sex),
Behavior by behavior {
    }
```

最后介绍外部正常构造野禽类的实例，注意代理类的入参要传送具体的行为对象，之后这只野禽实例就能很自如地按设定好的行为来飞呀、游呀、跑呀。下面是外部调用野禽类 WildFowl 的具体行为代码例子：

```
        btn_delegate_behavior.setOnClickListener {
            var fowl = when (count++%6) {
                //把代理类作为输入参数来创建实例
                0 -> WildFowl("老鹰", Bird.MALE, BehaviorFly())
                //由于 sex 字段是个默认参数，因此可通过命名参数给 behavior 赋值
                1 -> WildFowl("凤凰", behavior=BehaviorFly())
                2 -> WildFowl("大雁", Bird.FEMALE, BehaviorSwim())
                3 -> WildFowl("企鹅", behavior=BehaviorSwim())
                4 -> WildFowl("鸵鸟", Bird.MALE, BehaviorRun())
                else -> WildFowl("鹂鹊", behavior=BehaviorRun())
            }
            var action = when (count%11) {
                in 0..3 -> fowl.fly()
                4,7,10 -> fowl.swim()
                else -> fowl.run()
            }
            tv_class_inherit.text = "${fowl.name}: $action"
        }
```

以上接口代理（或称类代理）代码的运行结果如图 5-14～图 5-19 所示，其中图 5-14 表示老鹰

的飞翔行为，图 5-15 表示凤凰的游泳行为，图 5-16 表示大雁的飞翔行为，图 5-17 表示企鹅的游泳行为，图 5-18 表示鸵鸟的飞翔效果，图 5-19 表示鸸鹋的奔跑效果。



图 5-14　代理老鹰的飞翔行为

grammar
类继承信息：
凤凰：落水凤凰不如鸡

图 5-15　代理凤凰的游泳行为

grammar
类继承信息：
大雁：看情况，大雁能展翅高飞，企鹅却欲飞还休

图 5-16　代理大雁的飞翔行为

grammar
类继承信息：
企鹅：怡然戏水

图 5-17　代理企鹅的游泳行为

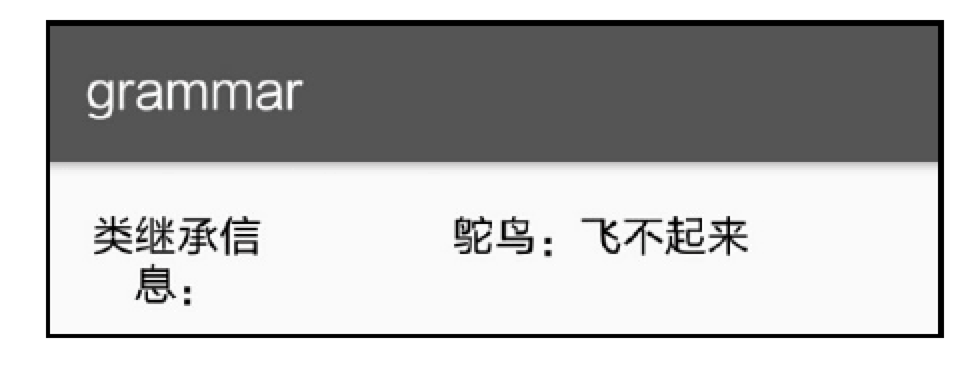


图 5-18　代理鸵鸟的飞翔行为

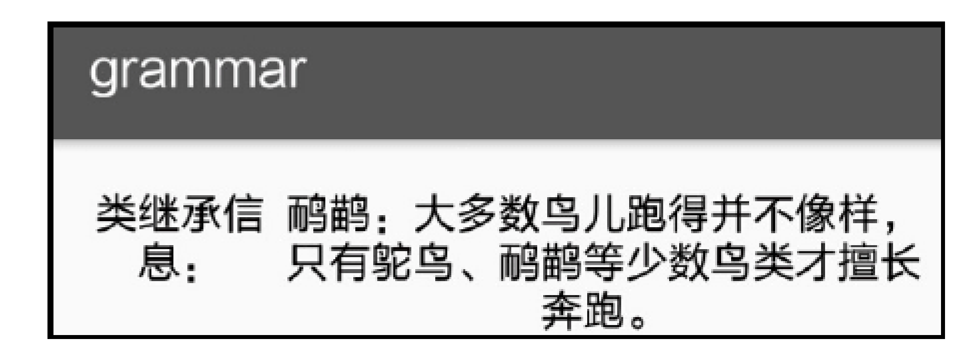


图 5-19　代理鸸鹋的奔跑行为

总结一下，Kotlin 的类继承与 Java 相比有所不同，主要体现在以下几点：

（1）Kotlin 的类默认不可被继承，若需继承，则要添加 open 声明；而 Java 的类默认是允许被继承的，只有添加 final 声明才表示不能被继承。

（2）Kotlin 除了常规的三个开放性修饰符 public、protected、private 外，另外增加了修饰符 internal，表示只对本模块开放。

（3）Java 的类继承关键字 extends 以及接口实现关键字 implement 在 Kotlin 中都被冒号所取代。

（4）Kotlin 允许在接口内部实现某个方法，而 Java 接口的内部方法只能是抽象方法。

（5）Kotlin 引入了接口代理（类代理）的概念，而 Java 不存在代理的写法。

5.4　几种特殊类

5.3 节介绍了 Kotlin 的几种开放性修饰符以及如何从基类派生出子类，其中提到了被 abstract 修饰的抽象类。除了与 Java 共有的抽象类外，Kotlin 还新增了好几种特殊类，这些特殊类分别适应不同的使用场景，极大地方便了开发者的编码工作。下面就来看看 Kotlin 在特殊类方面究竟提供了哪些独门秘笈。

5.4.1 嵌套类

一个类可以在单独的代码文件中定义，也可以在另一个类内部定义，后一种情况叫作嵌套类，即 A 类嵌套在 B 类之中。乍看过去，这个嵌套类的定义似乎与 Java 的嵌套类是一样的，但其实有所差别。Java 的嵌套类允许访问外部类的成员，而 Kotlin 的嵌套类不允许访问外部类的成员。倘若 Kotlin 的嵌套类内部强行访问外部类的成员，则编译器会报错“Unresolved reference: ***”，意思是找不到这个东西。

下面是 Kotlin 在外部类中定义嵌套类的代码例子：

```
class Tree(var treeName:String) {
    //在类内部再定义一个类，这个新类称作嵌套类
    class Flower (var flowerName:String) {
        fun getName():String {
            return "这是一朵$flowerName"
            //普通的嵌套类不能访问外部类的成员，如 treeName
            //否则编译器报错“Unresolved reference: ***”
            //return "这是${treeName}上的一朵$flowerName"
        }
    }
}
```

调用嵌套类时，得在嵌套类的类名前面添加外部类的类名，相当于把这个嵌套类作为外部类的静态对象使用。嵌套类 Flower 的调用代码如下所示：

```
    btn_class_nest.setOnClickListener {
        //使用嵌套类时，只能引用外部类的类名，不能调用外部类的构造函数
        val peachBlossom = Tree.Flower("桃花");
        tv_class_secret.text = peachBlossom.getName()
    }
```

因为嵌套类无法访问外部类的成员，所以其方法只能返回自身的信息，该例子中的嵌套类调用结果如图 5-20 所示，只看到了花儿类的花朵名称。



图 5-20 嵌套类的演示界面

5.4.2 内部类

既然 Kotlin 限制了嵌套类不能访问外部类的成员，那还有什么办法可以实现此功能呢？针对该问题，Kotlin 另外增加了关键字 inner 表示内部，把 inner 加在嵌套类的 class 前面，于是嵌套类华丽地转变为了内部类，这个内部类比起嵌套类的好处是能够访问外部类的成员。所以，Kotlin 的内部类就相当于 Java 的嵌套类，而 Kotlin 的嵌套类则是加了访问限制的内部类。

仍旧利用前面演示嵌套类的树木类 Tree，给它补充内部类 Fruit 的定义，具体代码如下所示：

```
class Tree(var treeName:String) {
    //在类内部再定义一个类，这个新类称作嵌套类
    class Flower (var flowerName:String) {
        fun getName():String {
            return "这是一朵$flowerName"
            //普通的嵌套类不能访问外部类的成员，如 treeName
            //否则编译器报错“Unresolved reference: ***”
            //return "这是${treeName}上的一朵$flowerName"
        }
    }

    //嵌套类加上 inner 前缀，就成为内部类
    inner class Fruit (var fruitName:String) {
        fun getName():String {
            //只有声明为内部类（添加了关键字 inner），才能访问外部类的成员
            return "这是${treeName}长出来的$fruitName"
        }
    }
}
```

调用内部类时，要先实例化外部类，再通过外部类的实例调用内部类的构造函数，也就是把内部类作为外部类的一个成员对象来使用，这与成员属性、成员方法的调用方法类似。外部调用内部类 Fruit 的代码如下所示：

```
    btn_class_inner.setOnClickListener {
        //使用内部类时，必须调用外部类的构造函数，否则编译器会报错
        val peach = Tree("桃树").Fruit("桃子");
        tv_class_secret.text = peach.getName()
    }
```

调用内部类 Fruit 的演示结果如图 5-21 所示，此时水果类的返回信息不但包含自身的果实名称，而且包含树木类的树木名称。



图 5-21　内部类的演示界面

5.4.3　枚举类

Java 有一种枚举类型，它采用关键字 enum 来表达，其内部定义了一系列名称，通过有意义的名字比 0、1、2 这些数字能够更有效地表达语义。下面是一个 Java 定义枚举类型的代码例子：

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER }
```

上面的枚举类型定义代码看起来仿佛是一种新的数据类型，特别像枚举数组。可是枚举类型实际上是一种类，开发者在代码中创建 enum 类型时，Java 编译器会自动生成一个对应的类，并且该类继承自 java.lang.Enum。因此，Kotlin 拨乱反正，摒弃了“枚举类型”那种模糊不清的说法，转而采取“枚举类”这种正本清源的提法。具体到编码上，是将关键字 enum 作为 class 的修饰符，使之名正言顺地成为一种类——枚举类。

按此思路将前面 Java 的枚举类型 Season 改写为 Kotlin 的枚举类，改写后的枚举类代码如下所示：

```
enum class SeasonType {
    SPRING, SUMMER, AUTUMN, WINTER
}
```

枚举类内部的枚举变量除了可以直接拿来赋值之外，还可以通过枚举值的几个属性获得对应的信息，例如 ordinal 属性用于获取该枚举值的序号，name 属性用于获取该枚举值的名称。枚举变量本质上还是该类的一个实例，所以如果枚举类存在构造函数，枚举变量也必须调用对应的构造函数。这样做的好处是，每个枚举值不但携带唯一的名称，还可以拥有更加个性化的特征描述。

比如下面的枚举类 SeasonName，其代码通过构造函数能够给枚举值赋予更加丰富的含义：

```
enum class SeasonName (val seasonName:String) {
    SPRING("春天"),
    SUMMER("夏天"),
    AUTUMN("秋天"),
    WINTER("冬天")
}
```

下面的代码分别演示如何使用枚举类 SeasonType 和 SeasonName：

```
    btn_class_enum.setOnClickListener {
        if (count%2 == 0) {
            //ordinal 表示枚举类型的序号，name 表示枚举类型的名称
            tv_class_secret.text = when (count++%4) {
                SeasonType.SPRING.ordinal -> SeasonType.SPRING.name
                SeasonType.SUMMER.ordinal -> SeasonType.SUMMER.name
                SeasonType.AUTUMN.ordinal -> SeasonType.AUTUMN.name
                SeasonType.WINTER.ordinal -> SeasonType.WINTER.name
                else -> "未知"
            }
        } else {
            tv_class_secret.text = when (count++%4) {
                //使用自定义属性 seasonName 表示更个性化的描述
                SeasonName.SPRING.ordinal -> SeasonName.SPRING.seasonName
                SeasonName.SUMMER.ordinal -> SeasonName.SUMMER.seasonName
                SeasonName.AUTUMN.ordinal -> SeasonName.AUTUMN.seasonName
                SeasonName.WINTER.ordinal -> SeasonName.WINTER.seasonName
                else -> "未知"
                //枚举类的构造函数是给枚举类型使用的，外部不能直接调用枚举类的构造函数
```

```
                //else -> SeasonName("未知").name
            }
        }
    }
```

5.4.4　密封类

5.4.3 小节演示外部代码判断枚举值的时候，when 语句末尾例行公事加了 else 分支。可是枚举类 SeasonType 内部一共有 4 个枚举变量，照理 when 语句有 4 个分支就行了，最后的 else 分支纯粹是多此一举。出现此情况的缘故是，when 语句不晓得 SeasonType 有 4 种枚举值，因此以防万一，必须要有 else 分支，除非编译器认为现有的几个分支已经足够。

为解决枚举值判断的多余分支问题，Kotlin 提出了“密封类”的概念，密封类就像是一种更加严格的枚举类，它内部有且仅有自身的实例对象，所以是一个有限的自身实例集合。或者说，密封类采用了嵌套类的手段，它的嵌套类全部由自身派生而来，仿佛一个家谱，明明白白地列出某人有长子、次子、三子、幺子。定义密封类的时候，需要在该类的 class 前面加上关键字 sealed 作为标记。

还是以 5.4.3 小节的四季为例，对应的密封类定义代码例子如下所示：

```
sealed class SeasonSealed {
    //密封类内部的每个嵌套类都必须继承该类
    class Spring (var name:String) : SeasonSealed()
    class Summer (var name:String) : SeasonSealed()
    class Autumn (var name:String) : SeasonSealed()
    class Winter (var name:String) : SeasonSealed()
}
```

这下有了密封类，外部使用 when 语句便无须指定 else 分支了。下面是判断密封类对象的代码例子：

```
    btn_class_sealed.setOnClickListener {
        var season = when (count++%4) {
            0 -> SeasonSealed.Spring("春天")
            1 -> SeasonSealed.Summer("夏天")
            2 -> SeasonSealed.Autumn("秋天")
            else -> SeasonSealed.Winter("冬天")
        }
        //密封类是一种严格的枚举类，它的值是一个有限的集合
        //密封类确保条件分支覆盖了所有的枚举类型，因此不再需要 else 分支
        tv_class_secret.text = when (season) {
            is SeasonSealed.Spring -> season.name
            is SeasonSealed.Summer -> season.name
            is SeasonSealed.Autumn -> season.name
            is SeasonSealed.Winter -> season.name
        }
    }
```

5.4.5 数据类

在 Android 开发中，免不了经常定义一些存放数据的实体类，比如用户信息、商品信息等，每逢定义实体类之时，开发者基本要手工完成以下编码工作：

（1）定义实体类的每个字段，以及对字段进行初始赋值的构造函数。

（2）定义每个字段的 get/set 方法。

（3）在判断两个数据对象是否相等时，通常每个字段都要比较一遍。

（4）在复制数据对象时，如果想另外修改某几个字段的值，得再补充对应数量的赋值语句。

（5）在调试程序时，为获知数据对象里保存的字段值，得手工把每个字段值都打印出来。

如此折腾一番，仅仅是定义一个实体类，开发者就必须完成这些烦琐的任务。然而这些任务其实毫无技术含量可言，假设每天都在周而复始地敲实体类的相关编码，毫无疑问跟工地上搬砖的民工差不多，活生生把程序员弄成一个拼时间、拼体力的职业。鉴于此，Kotlin 再次不负众望地推出了名为“数据类”的大兵器，直接戳中程序员事多、腰酸、睡眠少的痛点，极大程度上将程序员从无涯苦海中拯救出来。

数据类说神秘也不神秘，它的类定义代码极其简单，只要开发者在 class 前面增加关键字“data”，并声明拥有完整输入参数的构造函数，即可无缝实现以下功能：

（1）自动声明与构造函数入参同名的属性字段。

（2）自动实现每个属性字段的 get/set 方法。

（3）自动提供 equals 方法，用于比较两个数据对象是否相等。

（4）自动提供 copy 方法，允许完整复制某个数据对象，也可在复制后单独修改某几个字段的值。

（5）自动提供 toString 方法，用于打印数据对象中保存的所有字段值。

功能如此强大的数据类，犹如步枪界的 AK47，持有该款自动步枪的战士无疑战斗力倍增。见识了数据类的深厚功力，再来看看它的类代码是怎么定义的：

```
    //数据类必须有主构造函数，且至少有一个输入参数
    //并且要声明与输入参数同名的属性，即输入参数前面添加关键字 val 或者 var
    //数据类不能是基类也不能是子类，不能是抽象类，也不能是内部类，更不能是密封类
    data class Plant(var name:String, var stem:String, var leaf:String, var
flower:String, var fruit:String, var seed:String) {
    }
```

想不到吧，原来数据类的定义代码竟然如此简单，当真是此时无招胜有招。当然，为了达到这个代码精简的效果，数据类也得遵循几个规则，或者说是约束条件，毕竟不以规矩不成方圆，正如类定义代码所注释的那样：

（1）数据类必须有主构造函数，且至少有一个输入参数，因为它的属性字段要跟输入参数一一对应，如果没有属性字段，这个数据类保存不了数据，也就失去存在的意义。

（2）主构造函数的输入参数前面必须添加关键字 val 或者 var，这保证每个入参都会自动声明同名的属性字段。

（3）数据类有自己的一套行事规则，所以它只能是个独立的类，不能是其他类型的类，否则不同规则之间会产生冲突。

现在利用上面定义好的数据类——植物类 Plant，演示看看外部如何操作数据类。下面是外部调用植物类的具体代码：

```
var lotus = Plant("莲", "莲藕", "莲叶", "莲花", "莲蓬", "莲子")
//数据类的 copy 方法不带参数，表示复制一模一样的对象
var lotus2 = lotus.copy()
btn_class_data.setOnClickListener {
    lotus2 = when (count++%2) {
        //copy 方法带参数，表示指定参数另外赋值
        0 -> lotus.copy(flower="荷花")
        else -> lotus.copy(flower="莲花")
    }
    //数据类自带 equals 方法，用于判断两个对象是否一样
    var result = if (lotus2.equals(lotus)) "相等" else "不等"
    tv_class_secret.text = "两个植物的比较结果是${result}\n" +
            "第一个植物的描述是${lotus.toString()}\n" +
            "第二个植物的描述是${lotus2.toString()}"
}
```

由此可见，上述代码一口气调用了 Plant 对象的 copy、equals、toString 等方法，并且这些方法都是数据类自动提供的，实实在在提高了开发者的编码效率。演示代码的运行结果如图 5-22 和图 5-23 所示，其中图 5-22 所示为两个实例都是莲花时的比较结果，图 5-23 所示为一个是荷花一个是莲花时的比较结果。

grammar

执行结果：　两个植物的比较结果是相等
第一个植物的描述是
Plant(name=莲, stem=莲藕, leaf=莲叶, flower=莲花, fruit=莲蓬, seed=莲子)
第二个植物的描述是
Plant(name=莲, stem=莲藕, leaf=莲叶, flower=莲花, fruit=莲蓬, seed=莲子)

图 5-22　修改复制前的数据类信息

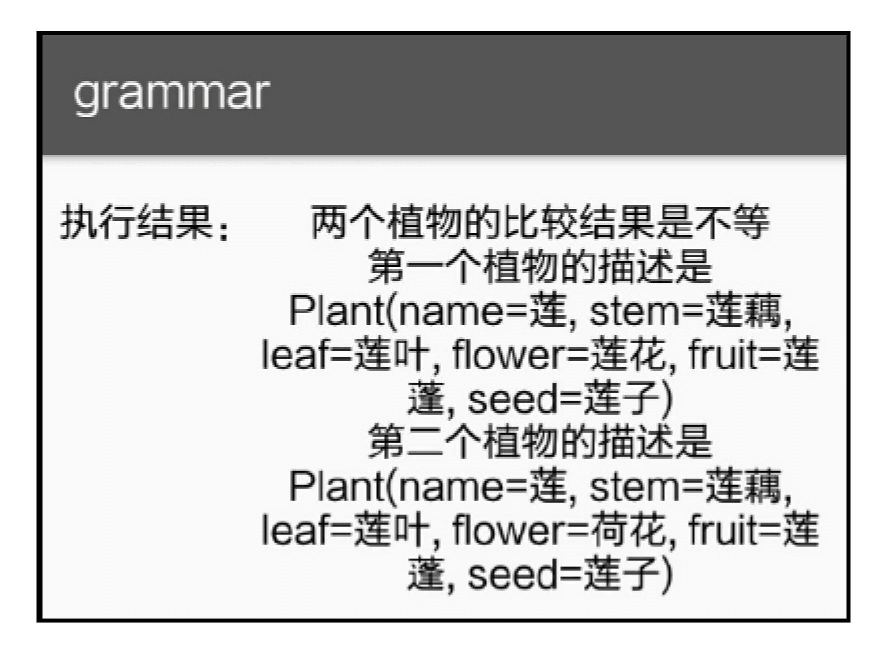


图 5-23　修改复制后的数据类信息

5.4.6　模板类

在第 4 章的“4.3.1　泛型函数”一节提到泛型函数的用法，当时把泛型函数作为全局函数定义，从而在别的地方也能调用它。那么如果某个泛型函数在类内部定义，即变成了这个类的成员方法，又该如何定义它呢？这个问题在 Java 中是通过模板类（也叫作泛型类）来解决的，例如常见的容器类 ArrayList、HashMap 均是模板类，Android 开发中的异步任务 AsyncTask 也是模板类。

模板类的应用如此广泛，Kotlin 自然而然保留了它，并且写法与 Java 类似，一样在类名后面

补充形如“<T>”或者“<A, B>”这样的表达式，表示此处的参数类型待定，要等创建类实例时再确定具体的参数类型。待定的类型可以有一个，如 ArrayList<T>；可以有两个，如 HashMap<K,V>；也可以有三个或者更多，如 AsyncTask<Params, Progress, Result>。

举个例子，森林里有一条小河，小河的长度可能以数字形式输入（包括 Int、Long、Float、Double），也可能以字符串形式输入（String 类型）。如果输入的是阿拉伯数字，长度单位就采取“m”；如果输入的是中文数字，长度单位就采取“米”。按照以上需求编写名为 River 的模板类，具体的河流类定义代码如下：

```
//在类名后面添加“<T>”，表示这是一个模板类
class River<T> (var name:String, var length:T) {
    fun getInfo():String {
        var unit:String = when (length) {
            is String -> "米"
            //Int、Long、Float、Double 都是数字类型 Number
            is Number -> "m"
            else -> ""
        }
        return "${name}的长度是$length$unit。"
    }
}
```

外部调用模板类构造函数的时候，要在类名后面补充“<参数类型>”，从而动态指定实际的参数类型。不过正如声明变量那样，如果编译器能够根据初始值判断该变量的类型，就无须显式指定该变量的类型。模板类也存在类似的偷懒写法，若编译器根据输入参数就能知晓参数类型，则调用模板类的构造函数也不必显式指定参数类型。

以下是外部使用河流模板类的代码例子：

```
    btn_class_generic.setOnClickListener {
        var river = when (count++%4) {
            //模板类(泛型类)声明对象时，要在模板类的类名后面加上“<参数类型>”
            0 -> River<Int>("小溪", 100)
            //如果编译器根据输入参数就能知晓参数类型，也可直接省略“<参数类型>”
            1 -> River("瀑布", 99.9f)
            //当然保守起见，新手最好按规矩添加“<参数类型>”
            2 -> River<Double>("山涧", 50.5)
            //如果你已经是老手了，怎么方便怎么来，Kotlin 的设计初衷就是偷懒
            else -> River("大河", "一千")
        }
        tv_class_secret.text = river.getInfo()
    }
```

最后看看河流类 River 的几种调用情况，分别如图 5-24～图 5-27 所示，其中图 5-24 展示小溪的长度描述文字，图 5-25 展示瀑布的长度描述文字，图 5-26 展示山涧的长度描述文字，图 5-27 展示大河的长度描述文字。可见只有大河的长度单位是中文的“米”，其他河流的长度单位都是“m”。

grammar

执行结果：　小溪的长度是100m。

图 5-24　小溪的长度信息

grammar

执行结果：　瀑布的长度是99.9m。

图 5-25　瀑布的长度信息

grammar

执行结果：　山涧的长度是50.5m。

图 5-26　山涧的长度信息

grammar

执行结果：　大河的长度是一千米。

图 5-27　大河的长度信息

5.5　小　　结

本章介绍了 Kotlin 对类由基础到高级的定义及实现过程，包括如何运用类的几种构造方式、如何在类内部定义成员属性和成员方法、如何在类内部正确使用伴生对象、如何实现类的几种继承方式（包括普通类、抽象类、接口、代理等）、如何在不同场合选用合适的特殊类等。

通过本章的学习，读者应能掌握以下技能：

（1）学会类的简单定义以及主构造函数和二级构造函数的用法。

（2）学会在类内部定义成员属性和成员方法，并借助伴生对象定义静态属性和静态方法。

（3）学会几种开放性修饰符的用法，并掌握普通类继承以及抽象类、接口、接口代理的实现。

（4）掌握常见的几种特殊类的定义和调用，包括嵌套类、内部类、枚举类、密封类、数据类、模板类等。

第 6 章

Kotlin 使用简单控件

Android 中的视图分为两大类，一类是布局，另一类是控件。布局与控件的区别在于，布局本质上是个容器，内部还可以放其他视图（包括子布局和子控件）；而控件是个单一的实体，已经是最后一级，下面不能再挂其他视图。布局和控件是 Android 初学者经常接触的东西，本章就从基本的视图开始，详细介绍如何使用 Kotlin 操作这些简单的布局和控件。

6.1 使用按钮控件

前面几章在演示 Kotlin 语法的时候，多次使用 Button 按钮控件来触发某项动作。不要小看这个小小的按钮，里面可大有玄机，比方说按钮事件不止点击一种，还有长按、选中、取消选中等事件；再比如按钮事件的实现代码，又有匿名函数、内部类、接口实现等方式；另外，按钮家族除了常见的 Button 控件外，还有复选框 CheckBox、单选按钮 RadioButton 等其他特殊按钮。原来看似简单的按钮，竟然存在这么多的学问，赶紧来看看 Kotlin 是如何使用各种按钮控件的。

6.1.1 按钮 Button

Button 是 Android 常用的控件之一，事实上按钮也是各大平台通用的基本控件，无论打开一个电脑程序还是手机 App，都会遇到“确定”“取消”“注册”“登录”等按钮，这些按钮的用途也很直白，都是用户点击一下触发某项动作。比如下面这行 Kotlin 代码便是一个简单的按钮点击事件的例子：

```
btn_click.setOnClickListener { btn_click.text="您点了一下下" }
```

按钮的长按事件处理与点击事件大同小异，区别在于长按代码末尾多了返回 true，长按事件具体的 Kotlin 代码示例如下：

```
        btn_click_long.setOnLongClickListener { btn_click_long.text="您长按了一小会"; true }
```

上面的两种按钮事件代码其实是简化最彻底的表达形式。因为点击事件和长按事件本来存在输入参数，它们的入参是发生了点击和长按动作的视图对象，所以完整的事件处理代码应当保留视图对象这个输入参数。只不过由于多数情况用不到视图对象，因此在 Kotlin 中把冗余的视图入参给省略了。但是为了弄清楚按钮事件的来龙去脉，还是有必要观察一下它的本来面貌，接下来依次介绍按钮事件的三种 Kotlin 编码方式：匿名函数、内部类、接口实现。

1. 匿名函数方式

譬如，现在准备响应按钮的点击事件，在点击按钮的同时提示该按钮的名称，此时点击事件的内部代码就得获取视图对象的文本。下面是补充了视图入参的 Kotlin 代码例子：

```
        //点击事件第一种：匿名函数方式
        btn_click_anonymos.setOnClickListener { v ->
            //Kotlin 对变量进行类型转换的关键字是 as
            toast("您点击了控件：${(v as Button).text}")
        }
```

由此可见，点击事件的函数代码被符号“->”分成了两部分：前一部分的“v”表示发生了点击动作的视图入参，其类型为 View；后一部分则为处理点击事件的具体函数体代码。此处的函数体代码中还有两个值得注意的地方：

（1）因为视图 View 是基本的视图类型，并不存在文本属性，所以需要把这个视图对象的变量类型转换为按钮 Button，然后才能得到按钮对象的文本。Kotlin 中的类型转换是通过关键字 as 实现的，具体的转换格式形如“待转换的变量名称 as 转换后的类型名称”。

（2）由于待显示的字符串需要拼接按钮文本，因此需要通过字符串模板表达式“${***}”将按钮文本置入该字符串。

这下有了包含输入参数的点击事件代码，书写包含入参的长按事件代码即可依样画葫芦，完整的 Kotlin 长按代码示例如下：

```
        //点击事件第一种：匿名函数方式
        btn_click_anonymos.setOnLongClickListener { v ->
            //Kotlin 对变量进行类型转换的关键字是 as
            longToast("您长按了控件：${(v as Button).text}")
            true
        }
```

以上带入参的函数处理代码构成了按钮事件的第一种写法——匿名函数方式，当然也是编码最简洁的一种方式。

2. 内部类方式

匿名函数方式直接把事件代码写在 setOnClickListener 方法后面，如果代码不多倒还凑合，要是代码很多那就尴尬了，只一个方法调用就会占去大量篇幅，着实显得大腹便便。故而对于包含较

多行代码的事件处理，往往给它单独定义一个内部类，这样该事件的处理代码被完全封装在内部类之中，能够有效增强代码的可读性。

就前面的点击事件和长按事件而言，可以给它们分别定义对应的监听器内部类，下面便是点击监听器内部类以及长按监听器内部类的Kotlin定义代码例子：

```
    //点击事件第二种：内部类方式
    private inner class MyClickListener : View.OnClickListener {
        override fun onClick(v: View) {
            toast("您点击了控件：${(v as Button).text}")
        }
    }

    private inner class MyLongClickListener : View.OnLongClickListener {
        override fun onLongClick(v: View): Boolean {
            longToast("您长按了控件：${(v as Button).text}")
            return true
        }
    }
```

定义了事件处理的内部类之后，按钮控件在调用setOnClickListener方法或者调用setOnLongClickListener方法之时，即可直接传入相应内部类的对象实例，具体调用的Kotlin代码如下所示：

```
        //点击事件第二种：内部类方式
        btn_click_inner.setOnClickListener(MyClickListener())
        btn_click_inner.setOnLongClickListener(MyLongClickListener())
```

从上面的调用代码看到，方法内部的输入参数为内部类定义的监听器实例。既然对象实例可以多次构造，这也就意味着内部类方式允许被不同控件多次调用，因此很大程度上提高了事件处理代码的复用性。

3. 接口实现方式

内部类方式固然使事件代码更加灵活，可如果每个事件都定义新的内部类，要是某个页面上多个控件都需要监听对应的事件处理，该页面的活动代码势必得定义一大堆监听器内部类，仍会造成拥挤不堪的代码局面。所以处理监听事件的第三种方式——接口实现方式便应运而生，该方式让页面的Activity类实现事件监听器的接口，并重写监听器的接口方法，使得这些接口方法就像是Activity类的成员方法一样，并且可以毫无障碍地访问该Activity类的所有成员属性和成员方法。

接口的概念及其具体实现参见第5章的“5.3.4 接口”。下面是在Activity类采取接口方式实现点击事件和长按事件的代码例子：

```
class ButtonClickActivity : AppCompatActivity(), OnClickListener,
OnLongClickListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_button_click)
```

```
        //点击事件第三种：Activity 实现接口
        btn_click_interface.setOnClickListener(this)
        btn_click_interface.setOnLongClickListener(this)
    }

    //点击事件第三种：Activity 实现接口
    override fun onClick(v: View) {
        if (v.id == R.id.btn_click_interface) {
            toast("您点击了控件：${(v as Button).text}")
        }
    }

    override fun onLongClick(v: View): Boolean {
        if (v.id == R.id.btn_click_interface) {
            longToast("您长按了控件：${(v as Button).text}")
        }
        return true
    }
}
```

当然，接口实现方式更适用于事件代码较多、较复杂的情况。总之，处理按钮控件的点击和长按事件具体采取哪种 Kotlin 编码方式，还是由实际的业务情况来决定。

6.1.2　复选框 CheckBox

复选框用于检查有没有选中的控件，该控件要么是选中状态，要么是取消选中状态，因此常常用于判断“是否***”的场合，比如“是否记住密码”“是否同意条款”“是否选择全部”等情况。

在学习复选框的用法之前，先了解一下复合按钮 CompoundButton 的概念。在 Android 体系中，CompoundButton 类是抽象的复合按钮，因为是抽象类，所以并不能直接使用。实际开发中用的是它的几个派生类，主要有复选框 CheckBox、单选按钮 RadioButton 以及开关按钮 Switch，这些派生类均可使用 CompoundButton 的属性和方法。

Android 处理复合按钮的勾选状态有两个 Java 方法：setChecked 和 isChecked，其中 setChecked 方法用于设置按钮的勾选状态，而 isChecked 方法用于判断按钮是否勾选。但在 Kotlin 编码中，这两个方法被统一成 isChecked 属性，修改 isChecked 的属性值即为设置按钮的勾选状态，而获取 isChecked 的属性值即为判断按钮是否勾选。对于这种将两个方法合二为一的按钮状态属性，Kotlin 还有好些类似的情况，具体的对应方法与属性说明参见表 6-1。

表 6-1　两个状态方法合并为一个状态属性的 Kotlin 与 Java 对照

按钮控件的属性说明	Kotlin 的状态属性	Java 的状态获取与设置方法
是否勾选	isChecked	isChecked/setChecked
是否允许点击	isClickable	isClickable/setClickable
是否可用	isEnabled	isEnabled/setEnabled

（续表）

按钮控件的属性说明	Kotlin 的状态属性	Java 的状态获取与设置方法
是否获得焦点	isFocusable	isFocusable/setFocusable
是否按下	isPressed	isPressed/setPressed
是否允许长按	isLongClickable	isLongClickable/setLongClickable
是否选择	isSelected	isSelected/setSelected

复选框 CheckBox 是复合按钮的一个简单实现，点击复选框则勾选，再次点击则取消勾选。CheckBox 通过 setOnCheckedChangeListener 方法设置勾选监听器，下面是使用复选框自定义勾选监听器的 Kotlin 代码例子：

```
class CheckboxActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_checkbox)
        ck_select.isChecked = false  //默认是未选中状态
        ck_select.setOnCheckedChangeListener { buttonView, isChecked ->
          tv_select.text = "您${ if (isChecked) "勾选" else "取消勾选"}了复选框"
        }
    }
}
```

这里的复选框演示代码主要运用了匿名函数、简单分支的 if 语句以及字符串模板等 Kotlin 基本语法。对应的复选框演示效果如图 6-1 和图 6-2 所示，其中图 6-1 所示为勾选后的界面，图 6-2 所示为取消勾选后的界面。



图 6-1 勾选了复选框的效果

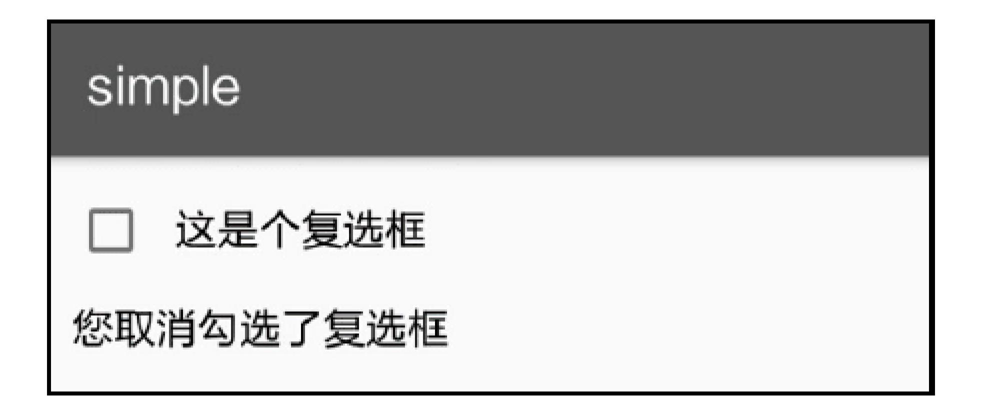


图 6-2 取消勾选之后的效果

6.1.3 单选按钮 RadioButton

单选按钮要在一组按钮中选择其中一项，并且不能多选，这要求有个容器确定这组按钮的范围，这个容器便是单选组 RadioGroup。单选组 RadioGroup 实质上是一个布局，同一组的 RadioButton 都要放在同一个 RadioGroup 节点之下。RadioGroup 拥有 orientation 属性，可指定下级控件的排列方向，该属性为 horizontal 时，单选按钮就在水平方向上排列；该属性为 vertical 时，单选按钮就在垂直方向上排列。并且 RadioGroup 下面除了 RadioButton 外，也可以挂载其他子控件，如 TextView、ImageView 等，这样看来，它就是一个特殊的线性布局，只不过多了一个管理单选按钮的功能。

单选按钮 RadioButton 默认是未选中状态，点击它则显示选中状态，但是再次点击并不会取消选中。只有点击同组的其他单选按钮，原来选中的单选按钮才会被取消选中。另外，单选按钮的选中事件一般不由 RadioButton 响应，而是由 RadioGroup 来响应。单选按钮的选中事件在实现的时候，首先写一个选中监听器实现接口 RadioGroup.OnCheckedChangeListener，然后调用 RadioGroup 对象的 setOnCheckedChangeListener 方法来注册该监听器。

下面是一个 RadioGroup 实现选中监听器的 Kotlin 代码例子：

```
class RadioButtonActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_radio_button)
        rg_sex.setOnCheckedChangeListener { group, checkedId ->
            tv_sex.text = when (checkedId) {
                R.id.rb_male -> "哇哦，你是个帅气的男孩"
                R.id.rb_female -> "哇哦，你是个漂亮的女孩"
                else -> ""
            }
        }
    }
}
```

由此可见，单选按钮的演示代码运用了匿名函数以及多路分支的 when 语句。对应的单选演示效果如图 6-3 和图 6-4 所示，其中图 6-3 所示为选中左侧按钮后的界面，图 6-4 所示为选中右侧按钮后的界面。

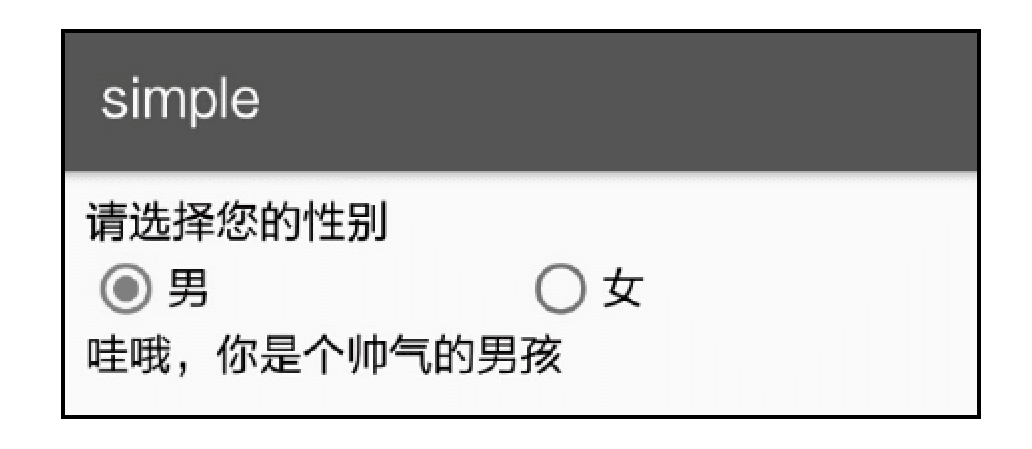


图 6-3　选中左侧按钮的效果



图 6-4　选中右侧按钮的效果

总结一下，本节介绍了常见的三种按钮控件的用法，并复习了 Kotlin 的一些基本语法知识，包括：

（1）Kotlin 的匿名函数用法。
（2）Kotlin 的内部类用法。
（3）Kotlin 的接口实现办法。
（4）Kotlin 的字符串模板写法。
（5）Kotlin 简单分支与多路分支的表达式。

另外，学习了 Kotlin 的类型转换关键字 as 的用法。

6.2 使用页面布局

布局视图有很多种，各自规定了内部下级视图的排列与对齐方式，包括线性布局 LinearLayout、相对布局 RelativeLayout、框架布局 FrameLayout 等，以及后起之秀约束布局 ConstraintLayout。本节对常用的两种布局——线性布局和相对布局进行说明，另外介绍使用约束布局的一些注意事项。

6.2.1 线性布局 LinearLayout

顾名思义，线性布局下面的子视图像是用一根线串起来，所以其内部视图的排列是有顺序的，要么从上到下垂直排列，要么从左到右水平排列。不过排列顺序只能指定一维方向的视图次序，可手机屏幕是一个二维的平面，这意味着还剩另一维方向需要指定视图的对齐方式。故而线性布局主要有以下两种属性设置方法。

（1）setOrientation：设置内部视图的排列方向。LinearLayout.HORIZONTAL 表示水平布局，LinearLayout.VERTICAL 表示垂直布局。

（2）setGravity：设置内部视图的对齐方式，对齐方式的取值说明见表 6-2。

表 6-2 对齐方式的取值说明

Gravity 类的对齐方式	说明
Gravity.LEFT	向左对齐
Gravity.RIGHT	向右对齐
Gravity.TOP	向上对齐
Gravity.BOTTOM	向下对齐
Gravity.CENTER	居中对齐

空白距离 margin 和间隔距离 padding 是另外两个常见的视图概念，margin 指的是当前视图与周围视图的距离，而 padding 指的是当前视图与内部视图的距离。这么说可能有些抽象，接下来还是做个实验，看看它们的显示效果到底有什么不同。下面是一个实验用的布局文件内容，通过背景色观察每个视图的区域范围：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <!-- 外层布局的背景色是蓝色 -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="300dp"
        android:background="#00aaff" >
```

```
        <!-- 中间布局的背景色是黄色 -->
        <LinearLayout
            android:id="@+id/ll_margin"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="#ffff99" >
            <!-- 内层视图的背景色是红色 -->
            <View
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:background="#ff0000" />
        </LinearLayout>
    </LinearLayout>
</LinearLayout>
```

与上述布局文件对应的页面 Kotlin 代码如下所示，目的是根据不同的按钮分别设置不同方向上的 margin 和 padding 数值：

```
//该页面用于演示 margin 和 padding 的区别
class LinearLayoutActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_linear_layout)
        //设置 ll_margin 内部视图的排列方式为水平排列
        ll_margin.orientation = LinearLayout.HORIZONTAL
        //设置 ll_margin 内部视图的对齐方式为居中对齐
        ll_margin.gravity = Gravity.CENTER
        btn_margin_vertical.setOnClickListener {
            //Kotlin 对变量进行类型转换的关键字是 as
            val params = ll_margin.layoutParams as LinearLayout.LayoutParams
            //setMargins 方法为设置该视图与外部视图的空白距离
            //此处设置左边和右边的 margin 空白距离为 50dp
            params.setMargins(0, dip(50), 0, dip(50))
            ll_margin.layoutParams = params
        }
        btn_margin_horizontal.setOnClickListener {
            val params = ll_margin.layoutParams as LinearLayout.LayoutParams
            //此处设置顶部和底部的 margin 空白距离为 50dp
            params.setMargins(dip(50), 0, dip(50), 0)
            ll_margin.layoutParams = params
        }
        //setPadding 方法为设置该视图与内部视图的间隔距离
        btn_padding_vertical.setOnClickListener {
            //此处设置左边和右边的 padding 间隔距离为 50dp
            ll_margin.setPadding(0, dip(50), 0, dip(50))
```

```
        }
        btn_padding_horizontal.setOnClickListener {
            //此处设置顶部和底部的 padding 间隔距离为 50dp
            ll_margin.setPadding(dip(50), 0, dip(50), 0)
        }
    }
}
```

演示过程中，一开始整个视图是红色的，先按下第一个按钮，视图顶部和底部各出现一段蓝色区域，如图 6-5 所示。接着按下第二个按钮，视图顶部和底部的蓝色消失，取而代之出现了左边和右边的蓝色区域，如图 6-6 所示。前面这两种情况表明 setMargins 方法控制着当前视图与外层视图的间隔距离。

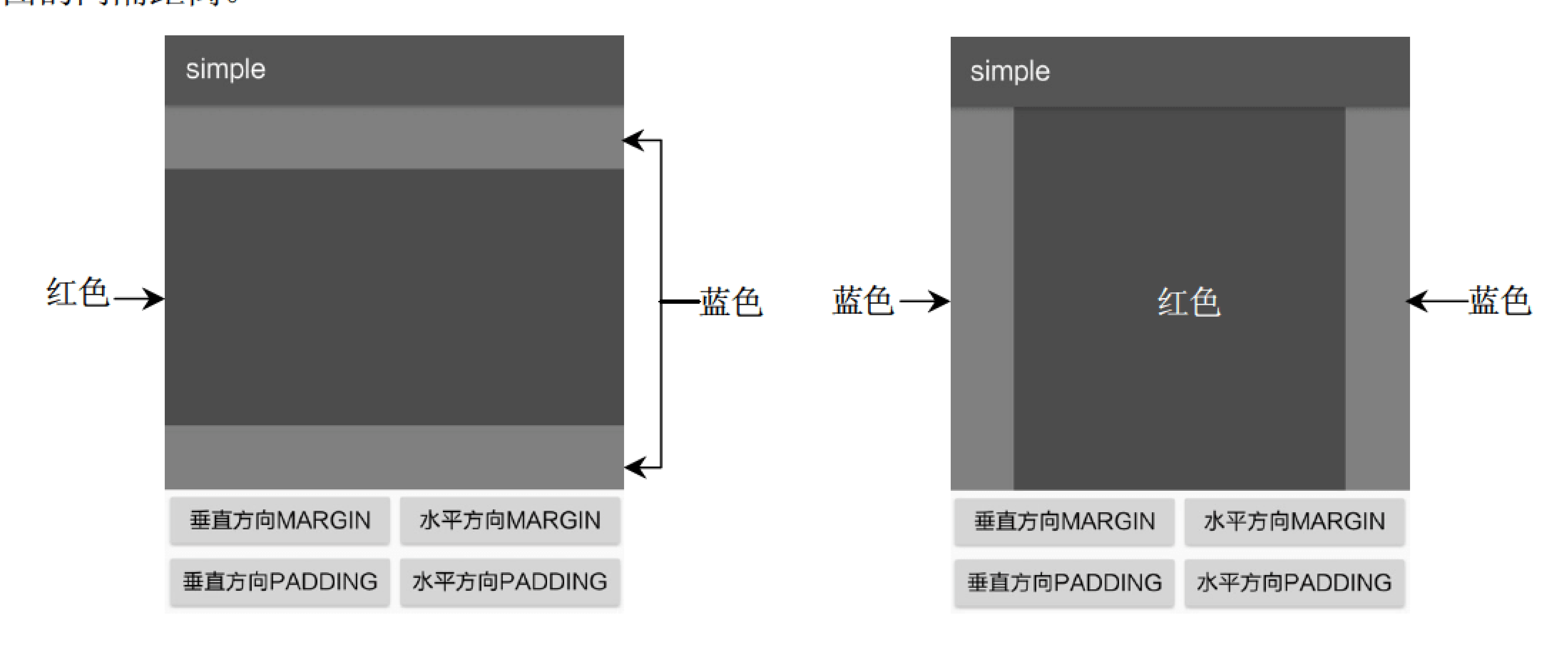


图 6-5　按下第一个按钮的线性布局效果　　图 6-6　按下第二个按钮的线性布局效果

再按下第三个按钮，此时左右两边的蓝色区域不变，但中间的红色区域上方和下方各出现一段黄色，如图 6-7 所示。最后按下第四个按钮，可见上方和下方的黄色消失，改到了在蓝色和红色区域之间出现黄色，如图 6-8 所示。后面两种情况表明 setPadding 方法控制着当前视图与内层视图的间隔距离。

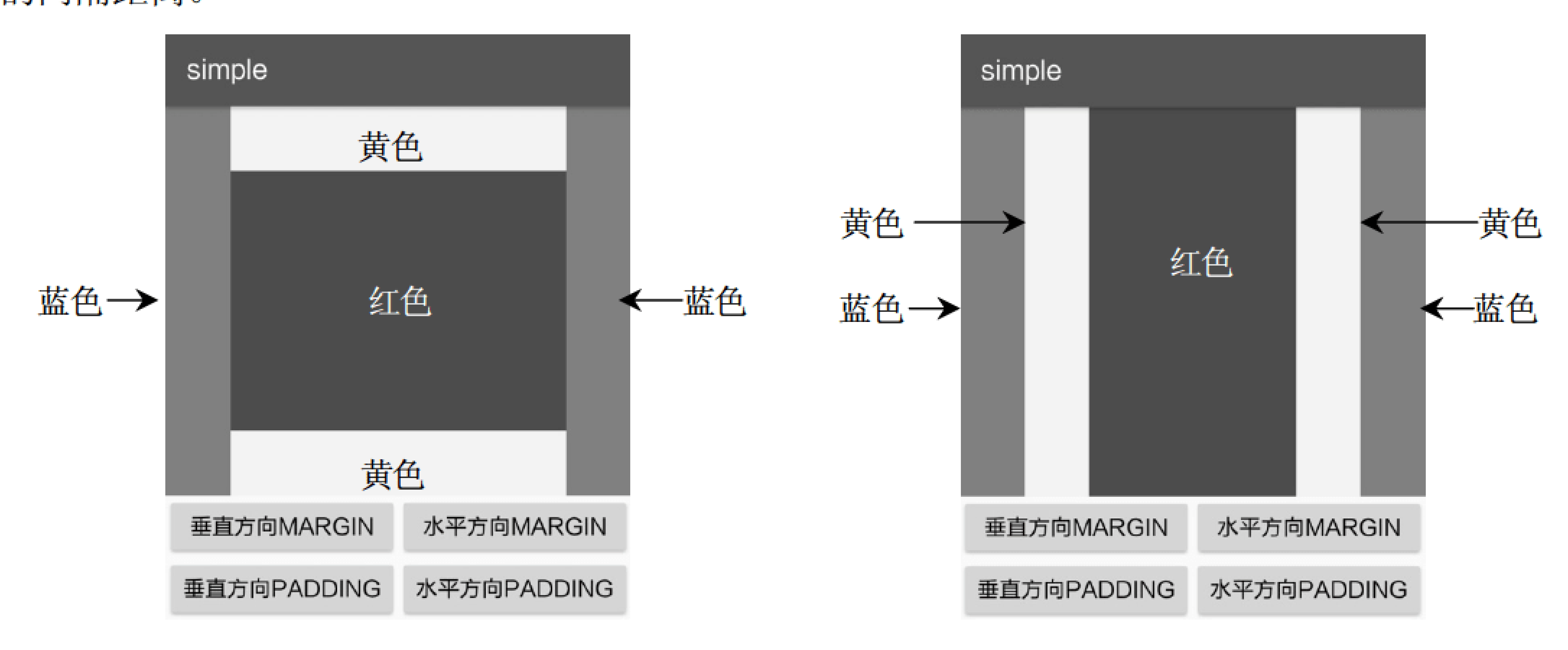


图 6-7　按下第三个按钮的线性布局效果　　图 6-8　按下第四个按钮的线性布局效果

依据前面的页面代码演示的例子，Kotlin 代码与 Java 代码的写法有以下三点区别：

（1）Kotlin 允许对属性 orientation 直接赋值，从而取代了 setOrientation 方法；类似的还有属性 gravity 取代了 setGravity 方法。

（2）Kotlin 使用关键字 as 进行变量的类型转换操作。

（3）Kolin 支持调用 dip 方法将 dip 数值转换为 px 数值，倘若由 Java 编码，则需开发者自己实现一个像素转换的工具类。下面是实现像素转换的 Java 工具类代码例子：

```
    public class Utils {
        //根据手机的分辨率从 dp 的单位转成为 px(像素)
        public static int dip2px(Context context, float dpValue) {
            final float scale = context.getResources().getDisplayMetrics()
.density;
            return (int) (dpValue * scale + 0.5f);
        }

        //根据手机的分辨率从 px(像素)的单位转成为 dp
        public static int px2dip(Context context, float pxValue) {
            final float scale = context.getResources().getDisplayMetrics()
.density;
            return (int) (pxValue / scale + 0.5f);
        }
    }
```

因为演示代码里的 dip 方法来自于 Kotlin 扩展的 Anko 库，所以需要在 Activity 代码头部加入下面一行导入语句：

```
    import org.jetbrains.anko.dip
```

既然用到了 Anko 库，自然要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
        compile "org.jetbrains.anko:anko-common:$anko_version"
```

Anko 库除了提供 dip 方法外，还提供了 sp、px2dip、px2sp、dimen 等像素单位的转换方法，具体的方法说明见表 6-3。

表 6-3 Anko 库的像素单位转换方法说明

Anko 库的像素单位转换方法	说明
dip	将 dip 单位的数值转换为以 px 为单位的数值
sp	将 sp 单位的数值转换为以 px 为单位的数值
px2dip	将 px 单位的数值转换为以 dip 为单位的数值
px2sp	将 px 单位的数值转换为以 sp 为单位的数值
dimen	将 dip 单位的数值转换为以 sp 为单位的数值

6.2.2 相对布局 RelativeLayout

由于线性布局的视图排列方式比较固定，既不能重叠显示也不能灵活布局，因此复杂一些的界面往往用到相对布局 RelativeLayout。相对布局内部的视图位置不依赖于排列规则，而依赖于指定的参照物，这个参照物可以是与该视图平级的视图，也可以是该视图的上级视图（上级视图即相对布局自身）。有了参照物之后，还得指定当前视图位于参照物的哪个方向，才能确定该视图的具体位置。

在代码中指定参照物及其所处方位，调用的是布局参数对象的 addRule 方法，方法格式形如“addRule(方位类型, 参照物的资源 ID)”。下面是一个给相对布局添加下级视图的 Kotlin 代码例子：

```
    //根据参照物与方位类型添加下级视图
    private fun addNewView(align: Int, referId: Int) {
        var v = View(this)
        v.setBackgroundColor(Color.GREEN)
        val rl_params = RelativeLayout.LayoutParams(100, 100)
        rl_params.addRule(align, referId)
        v.layoutParams = rl_params
        v.setOnLongClickListener { vv -> rl_content.removeView(vv); true}
        rl_content.addView(v)
    }
```

相对布局代码里的方位类型有多种取值，比如 RelativeLayout.LEFT_OF 表示位于指定视图的左边，RelativeLayout.ALIGN_RIGHT 表示与指定视图右侧对齐，RelativeLayout.CENTER_IN_PARENT 表示位于上级视图中央等。举个例子，要让某视图位于指定视图上方，并且与上级视图的左侧对齐，则调用 addRule 方法的 Kotlin 代码如下所示：

```
        rl_params.addRule(RelativeLayout.ABOVE, 指定视图的资源 ID)
        rl_params.addRule(RelativeLayout.ALIGN_PARENT_LEFT, 上级视图的资源 ID)
```

由此可见，常规的 addRule 调用代码有点冗长，因此 Kotlin 利用 Anko 库将相对位置的写法进行了简化，具体办法是引入扩展函数实现相对位置的设定，譬如 above 方法表示当前视图位于指定视图上方，而 alignParentLeft 方法表示当前视图与上级视图的左侧对齐。于是原来指定相对位置的 Kotlin 代码简化如下：

```
        rl_params.above(指定视图的资源 ID)
        rl_params.alignParentLeft()
```

因为这几个新方法都来自于 Anko 库，所以要在代码头部加入下面一行导入语句：

```
import org.jetbrains.anko.*
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
        compile "org.jetbrains.anko:anko-common:$anko_version"
```

除了 above 和 alignParentLeft 之外，Anko 还提供了其余的相对位置设定方法，它们与原来写法的对应关系说明见表 6-4。

表 6-4 相对位置的 Anko 方法与 RelativeLayout 类的对应关系

相对位置说明	Anko 库的相对位置	RelativeLayout 类的相对位置
位于指定视图左边	leftOf	LEFT_OF
与指定视图头部对齐	sameTop	ALIGN_TOP
位于指定视图上方	Above	ABOVE
与指定视图左侧对齐	sameLeft	ALIGN_LEFT
位于指定视图右边	rightOf	RIGHT_OF
与指定视图底部对齐	sameBottom	ALIGN_BOTTOM
位于指定视图下方	Below	BELOW
与指定视图右侧对齐	sameRight	ALIGN_RIGHT
位于上级视图中央	centerInParent	CENTER_IN_PARENT
与上级视图左侧对齐	alignParentLeft	ALIGN_PARENT_LEFT
位于上级视图垂直方向的中部	centerVertically	CENTER_VERTICAL
与上级视图顶部对齐	alignParentTop	ALIGN_PARENT_TOP
位于上级视图水平方向的中部	centerHorizontally	CENTER_HORIZONTAL
与上级视图右侧对齐	alignParentRight	ALIGN_PARENT_RIGHT
与上级视图底部对齐	alignParentBottom	ALIGN_PARENT_BOTTOM

6.2.3 约束布局 ConstraintLayout

约束布局 ConstraintLayout 是 Android Studio 2.2 开始推出的新布局，并从 Android Studio 2.3 开始成为默认布局文件的根布局，由此可见 Android 官方对其寄予厚望，那么约束布局究竟具备哪些激动人心的特性呢？

传统的布局如线性布局 LinearLayout、相对布局 RelativeLayout 等，若要描绘不规则的复杂界面，往往需要进行多重的布局嵌套，不但僵硬死板、缺乏灵活性，并且嵌套过多拖慢页面渲染速度。约束布局的出现正是为了解决这些问题，它兼顾灵活性和高效率，可以看作是相对布局的升级版，在很大程度上改善了 Android 的用户体验。开发者使用约束布局时，有多种手段往该布局内添加和拖动控件，既能像原型设计软件 AxureRP 那样在画板上任意拖曳控件，也能像传统布局那样在 XML 文件中调整控件布局，还能在代码中动态修改控件对象的位置状态。下面分别介绍约束布局的这几种使用方式。

1. 在画板上拖曳控件

设计师通过工具软件三两下就勾勒出界面原型，程序员却得一个控件一个控件地小心布局，并对控件位置不断微调以符合原型上的尺寸比例。Android 原先的这种界面手工编码方式一直为人所诟病，因为“所见即所得”才是界面编码的理想方式，比如 iOS 很早就在 Xcode 中集成了故事板，使得 iOS 程序员能够像设计师那样在画板上拖动控件，从而加快了界面编码的工作效率。所幸自从约束布局 ConstraintLayout 诞生之后，Android 程序员终于跟上时代步伐，也能在约束布局

内部随意拖曳控件，同时存在主从关系的控件之间，附庸控件会跟随目标控件一起移动，从而省却了界面微调的大量劳动。

画板上的约束布局控件拖动效果如图 6-9 和图 6-10 所示，其中图 6-9 所示为拖动前的画板界面，图 6-10 所示为拖动后的画板界面。



图 6-9　拖动前的约束布局面板

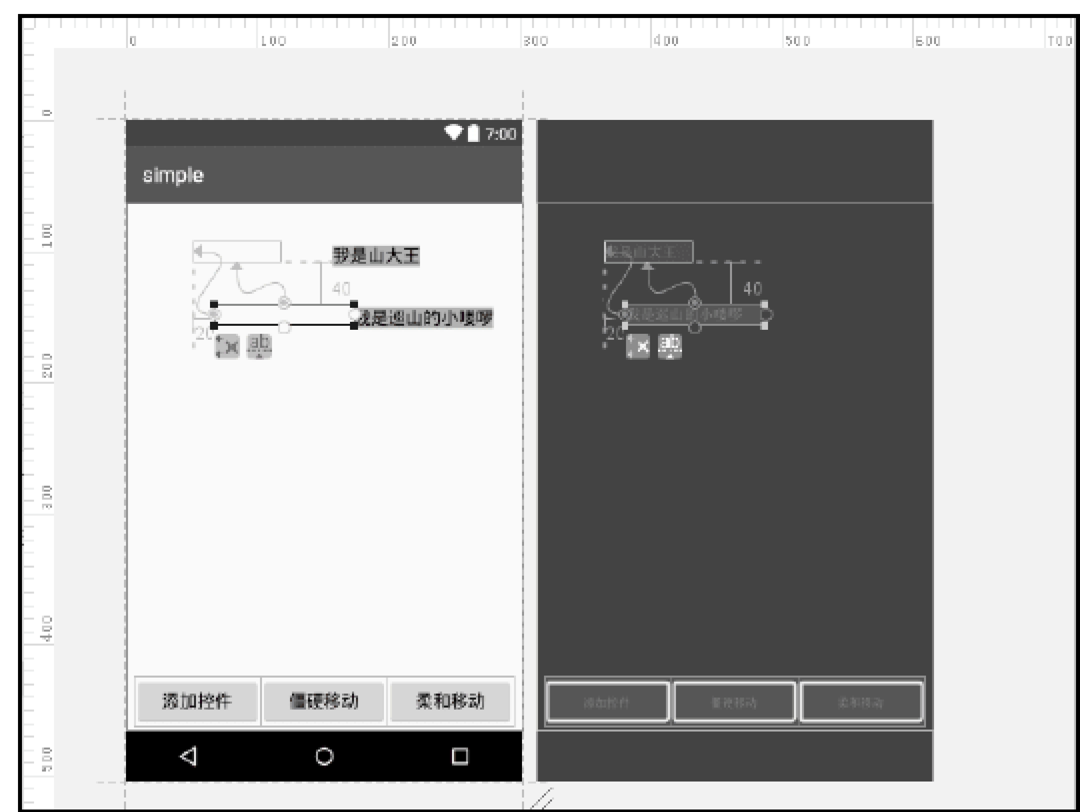


图 6-10　拖动后的约束布局面板

2. 在 XML 文件中调整控件布局

传统布局如线性布局、相对布局基本是在 XML 文件中手工添加控件节点，约束布局当然也允许在布局文件中指定控件的相对位置，这跟相对布局内部的控件位置调整类似，只不过用来表示位置的属性换了个名字罢了。与控制方位有关的属性说明如下所示：

- layout_constraintTop_toTopOf: 该控件的顶部与另一个控件的顶部对齐。
- layout_constraintTop_toBottompOf: 该控件的顶部与另一个控件的底部对齐。
- layout_constraintBottom_toTopOf : 该控件的底部与另一个控件的顶部对齐。
- layout_constraintBottom_toBottomOf: 该控件的底部与另一个控件的底部对齐。
- layout_constraintLeft_toLeftOf : 该控件的左侧与另一个控件的左侧对齐。
- layout_constraintLeft_toRightOf : 该控件的左侧与另一个控件的右侧对齐。
- layout_constraintRight_toLeftOf : 该控件的右侧与另一个控件的左侧对齐。
- layout_constraintRight_toRightOf : 该控件的右侧与另一个控件的右侧对齐。

下面是一个运用约束布局的 XML 文件例子：

```
    <android.support.constraint.ConstraintLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:id="@+id/cl_content"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/tv_first"
            android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginTop="40dp"
        app:layout_constraintLeft_toLeftOf="parent"
        android:layout_marginLeft="200dp"
        android:background="@color/blue"
        android:text="我是山大王"
        android:textSize="17sp"
        android:textColor="@color/black" />

    <TextView
        android:id="@+id/tv_second"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="40dp"
        app:layout_constraintTop_toBottomOf="@+id/tv_first"
        android:layout_marginLeft="20dp"
        app:layout_constraintLeft_toLeftOf="@+id/tv_first"
        android:background="@color/blue"
        android:text="我是巡山的小喽啰"
        android:textSize="17sp"
        android:textColor="@color/black" />
</android.support.constraint.ConstraintLayout>
```

与该布局文件对应的效果界面如图 6-11 所示，可见第二个文本视图的位置由第一个文本视图的位置决定。

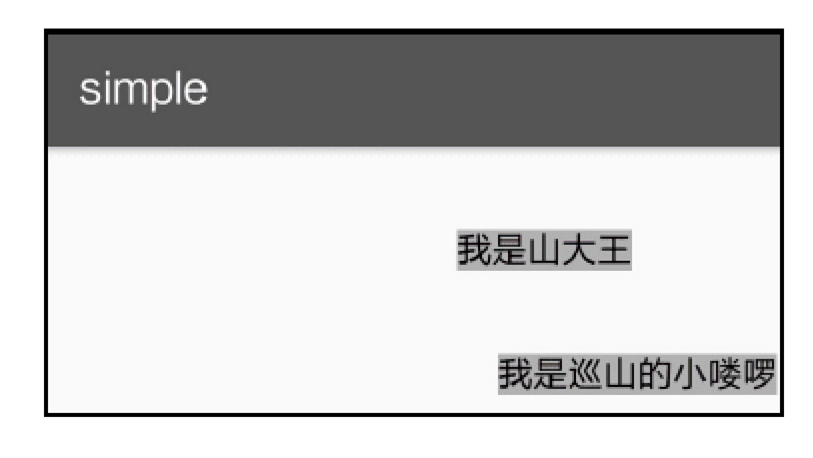


图 6-11　在 XML 布局中添加约束布局

3. 在代码中添加控件

若要利用代码给约束布局动态添加控件，则可照常调用 addView 方法。不同之处在于，新控件的布局参数必须使用约束布局的布局参数，即 ConstraintLayout.LayoutParams，该参数通过 setMargins/setMarginStart/setMarginEnd 方法设置新控件与周围控件的间距。至于新控件与周围控件的位置约束关系，则参照 ConstraintLayout.LayoutParams 的下列属性说明。

- topToTop：当前控件的顶部与指定 ID 的控件顶部对齐。
- topToBottom：当前控件的顶部与指定 ID 的控件底部对齐。
- bottomToTop：当前控件的底部与指定 ID 的控件顶部对齐。
- bottomToBottom：当前控件的底部与指定 ID 的控件底部对齐。
- startToStart：当前控件的左侧与指定 ID 的控件左侧对齐。
- startToEnd：当前控件的左侧与指定 ID 的控件右侧对齐。
- endToStart：当前控件的右侧与指定 ID 的控件左侧对齐。
- endToEnd：当前控件的右侧与指定 ID 的控件右侧对齐。

下面是在约束布局中添加新控件的 Kotlin 代码例子：

```
    private fun addNewView() {
        val tv = TextView(this)
        tv.text = "长按删除该文本"
        val container = ConstraintLayout.LayoutParams(
                ConstraintLayout.LayoutParams.WRAP_CONTENT,
                ConstraintLayout.LayoutParams.WRAP_CONTENT
        )
        //设置控件左侧与另一个控件的左侧对齐
        //水平方向上只能使用 start 和 end，因为 left 和 right 可能无法奏效
        container.startToStart = lastViewId
        //设置控件顶部与另一个控件的底部对齐
        container.topToBottom = lastViewId
        container.topMargin = dip(30)
        //左侧间距要使用 Start，不能用 Left，因为 set.applyTo 方法会清空 Left 的间距。
marginStart 需要 API17 支持
        container.marginStart = dip(10)
        tv.layoutParams = container
        tv.setOnLongClickListener { vv -> cl_content.removeView(vv); true }
        lastViewId += 1000
        tv.id = lastViewId
        cl_content.addView(tv)
    }
```

添加新控件的效果如图 6-12 和图 6-13 所示，其中图 6-12 所示为添加左边的第一个文本视图后的界面，图 6-13 所示为添加左边的第二个文本视图后的界面。



图 6-12 代码添加第一个约束关系的 TextView

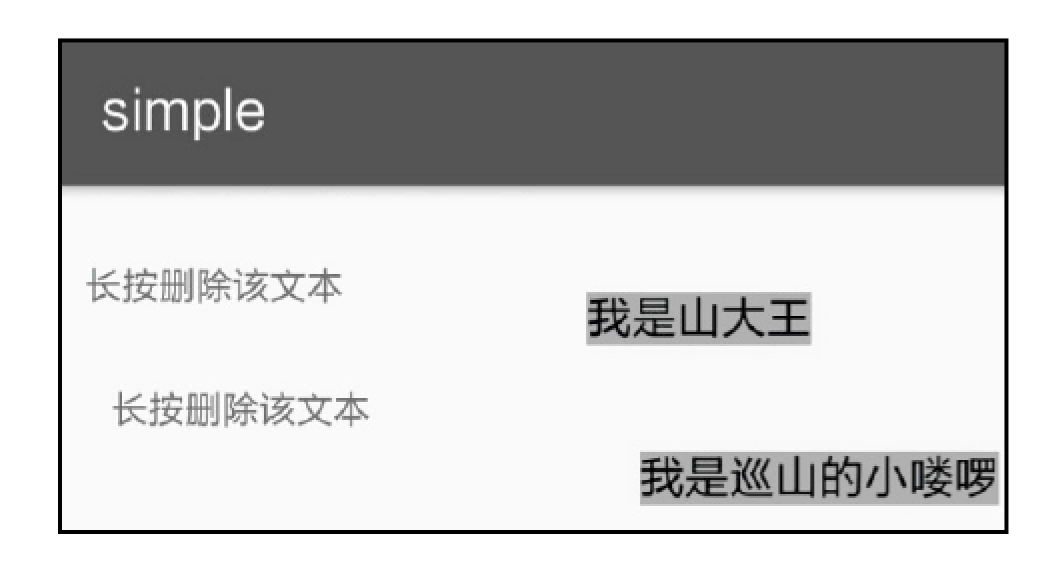


图 6-13 代码添加第二个约束关系的 TextView

4. 在代码中动态调整控件位置

有时根据用户在界面上的操作需要立即调整相关控件的显示位置，这要在代码中修改控件的位置参数。既然添加控件时可以通过布局参数指定控件位置，那么调整控件位置一样也可以通过布局参数来实现，基本流程依次为：先调用 getLayoutParams 方法获得当前的布局参数，再指定新的控件约束关系及间距，最后调用 setLayoutParams 启用新的布局参数。

可是按照传统的布局参数方式存在诸多不便之处，比如以下几点就很不合理：

（1）控件约束关系的目标指定与间距设定是分开的，其他人难以找到二者之间的对应 关系。

（2）setMargins 方法同时设置上下左右四个方向的间距，无法单独设置某个方向的间距。

（3）布局参数在启用时立即生效，也就是说控件位置一瞬间挪动，没有渐变的过程，让用户

觉得很突兀。控件位置的整个移动过程只有两个界面，分别如图 6-14 和图 6-15 所示，其中图 6-14 所示为移动之前的界面，图 6-15 所示为移动之后的界面。



图 6-14　约束控件移动之前的界面



图 6-15　约束控件移动之后的界面

为了改进以上几个问题，constraint-layout 开发包从 1.0.1 本版开始增加了新的约束设置类 ConstraintSet，该工具针对这几个问题分别给出了相应的解决方案：

（1）提供 connect 方法，一次性指定存在约束关系的两个控件以及它们的间距。

（2）提供 setMargin 方法，通过指定方向参数，从而允许单独设置上下左右某个方向的间距。

（3）提供渐变管理类 TransitionManager，以支持展示空间位置变化的切换动画。

下面是使用 ConstraintSet 修改控件位置的具体 Kotlin 代码：

```
    private fun moveView() {
        val margin = dip((if (isMoved) 200 else 20).toFloat())
        //需要下载最新的 constraint-layout 才能使用 ConstraintSet
        val set = ConstraintSet()
        //复制原有的约束关系
        set.clone(cl_content)
        //清空该控件的约束关系
        //set.clear(tv_first.getId());
        //设置该控件的约束宽度
        //set.constrainWidth(tv_first.getId(), ConstraintLayout.LayoutParams.
WRAP_CONTENT);
        //设置该控件的约束高度
        //set.constrainHeight(tv_first.getId(),ConstraintLayout.LayoutParams.
WRAP_CONTENT);
        //设置该控件的顶部约束关系与间距
        //set.connect(tv_first.getId(), ConstraintSet.TOP, cl_content.getId(),
ConstraintSet.BOTTOM, margin);
        //设置该控件的底部约束关系与间距
        //set.connect(tv_first.getId(), ConstraintSet.BOTTOM,
cl_content.getId(), ConstraintSet.BOTTOM, margin);
        //设置该控件的左侧约束关系与间距
        set.connect(tv_first.id, ConstraintSet.START, cl_content.id,
ConstraintSet.START, margin)
        //设置该控件的右侧约束关系与间距
```

```
            //set.connect(tv_first.getId(), ConstraintSet.END, cl_content.getId(),
ConstraintSet.END, margin);
            //LEFT 和 RIGHT 的 margin 不管用，只有 START 和 END 的 margin 才管用
            //set.setMargin(tv_init.getId(), ConstraintSet.START, 200);
            //启用新的约束关系
            set.applyTo(cl_content)
            isMoved = !isMoved
        }
```

为了能够显示位置变化的动画，在移动控件之前要先通过管理工具 TransitionManager 开启渐变动画效果。设定动画功能的 Kotlin 代码如下所示：

```
        btn_move_soft.setOnClickListener {
            //使用动画展示新旧约束关系的切换过程。若删掉这行，则不展示切换动画。该方法需要 API19
支持
            TransitionManager.beginDelayedTransition(cl_content)
            moveView()
        }
```

上述变更控件位置代码的对应效果如图 6-16 和图 6-17 所示，其中图 6-16 展示移动开始不久的界面，图 6-17 展示移动将要结束的界面，可见有了切换动画看起来就比较柔和了。

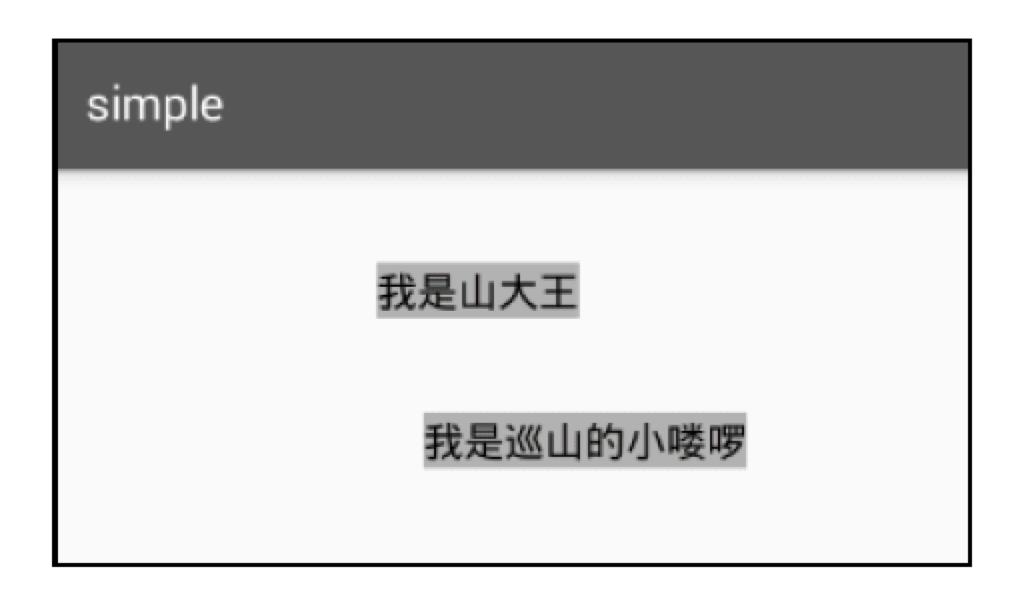


图 6-16　约束布局移动动画正在开始播放

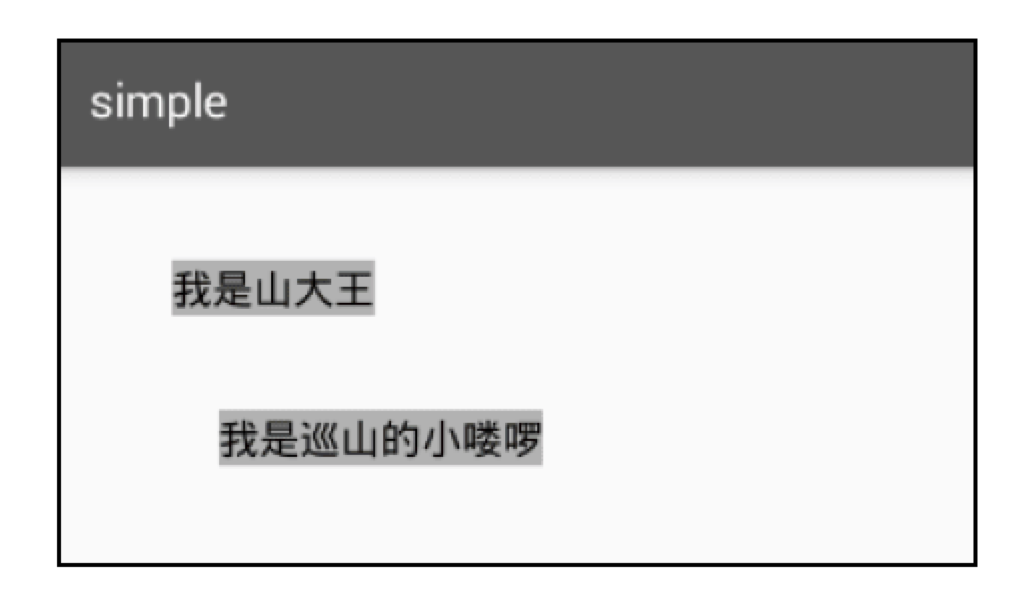


图 6-17　约束布局移动动画即将结束播放

6.3　使用图文控件

手机 App 的酷炫界面其实是由大量文本和图片以及各种特效堆砌出来的，这里面的基础控件无非就是文本视图与图像视图，前者专门用于显示文字，而后者专门用于显示图片，因而两者构成了多彩界面的基石。当然手机作为一种智能终端，需要方便接收用户的输入信息，这又用到了另一种基础控件——文本编辑框。编辑框与文本视图、图像视图一同组成 Android 三种常用的图文控件，接下来就对这三种图文控件分别进行介绍。

6.3.1　文本视图 TextView

6.2.3 小节介绍约束布局时，通过动态添加文本视图演示布局效果，当时只用到了 text 属性填写

文本内容。但是文本视图 TextView 并不仅限于显示简单的文本，还能用来展示某些特效文字效果，比如常见的跑马灯动画。当一行文本的内容太多，导致无法全部显示，但也不想分行展示时，只能让文字从左向右滚动显示，类似于跑马灯效果。像电视在播报突发新闻时，就经常在屏幕下方轮播消息文字，譬如“快讯：我国选手***在刚刚结束的**比赛中为中国代表团夺得第**枚金牌”等。

若要通过代码实现跑马灯滚动文字的特效，则需联合设置文本的多个属性值，包括省略方式 ellipsize 设定为 TextUtils.TruncateAt.MARQUEE，还要设定单行显示并获得焦点等。实现跑马灯效果的 Kotlin 代码例子如下所示：

```
    class TextMarqueeActivity : AppCompatActivity() {
        private var bPause = false

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_text_marquee)
            tv_marquee.text = "快讯：红色预警，超强台风“泰利”即将登陆，请居民关紧门窗、
备足粮油，做好防汛救灾准备！"
            tv_marquee.textSize = 17f
            tv_marquee.setTextColor(Color.BLACK)
            tv_marquee.setBackgroundColor(Color.WHITE)
            tv_marquee.gravity = Gravity.LEFT or Gravity.CENTER //左对齐且垂直居中
            tv_marquee.ellipsize = TextUtils.TruncateAt.MARQUEE //从右向左滚动的跑
马灯
            tv_marquee.setSingleLine(true) //跑马灯效果务必设置 SingleLine 单行显示
            tv_marquee.setOnClickListener {
                bPause = !bPause
                tv_marquee.isFocusable = if (bPause) false else true
                tv_marquee.isFocusableInTouchMode = if (bPause) false else true
            }
        }
    }
```

跑马灯滚动的效果界面如图 6-18 和图 6-19 所示，其中图 6-18 表示跑马灯文字在滚动之中，图 6-19 表示跑马灯文字停止滚动。

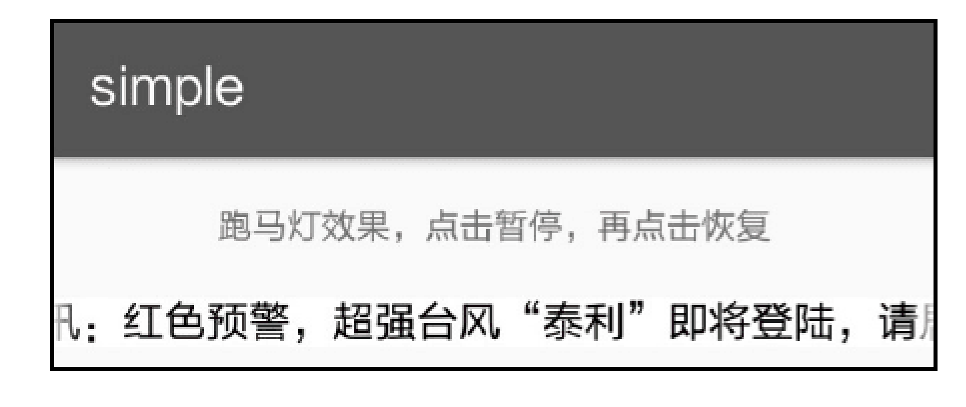


图 6-18　跑马灯文字正在滚动



图 6-19　跑马灯文字停止滚动

看过了跑马灯的效果图，再回头浏览实现该功能的 Kotlin 代码，发现 TextView 的部分属性允许直接赋值，而另一部分属性仍需通过方法设置。这些属性设置的 Kotlin 和 Java 实现方式对比见表 6-5。

表 6-5　文本视图属性设置的 Kotlin 和 Java 实现方式对比

文本视图的属性设置说明	Kotlin 的实现方式	Java 的实现方式
文本内容	text	setText
文本大小	textSize	setTextSize
文本颜色	setTextColor	setTextColor
背景颜色	setBackgroundColor	setBackgroundColor
对齐方式	gravity	setGravity
多余文本的省略方式，取值说明见表 6-6	ellipsize	setEllipsize
是否单行显示	setSingleLine	setSingleLine
是否获得焦点	isFocusable	setFocusable
是否在触摸时获得焦点	isFocusableInTouchMode	setFocusableInTouchMode

表 6-6　多余文本的省略方式

TruncateAt 类的省略方式	说明
TruncateAt.START	省略号在开头
TruncateAt.MIDDLE	省略号在中间
TruncateAt.END	省略号在末尾
TruncateAt.MARQUEE	跑马灯显示

除了以上的属性设置方式产生变动外，另外注意到文本对齐方式的赋值情况也有变化，原来 Java 设置文本对齐方式的代码是下面这样的：

```
tv_marquee.setGravity(Gravity.LEFT | Gravity.CENTER);
```

然而 Kotlin 对应的对齐设置代码却是以下格式：

```
tv_marquee.gravity = Gravity.LEFT or Gravity.CENTER
```

由此可见，对齐方式的或操作外在 Java 中采取竖线“|”表示，但在 Kotlin 中采取关键字“or”表示。不只是这个或运算，所有的位运算都被 Kotlin 定义了新的关键字，表 6-7 列举了常用的几种位运算符在 Kotlin 和 Java 中的展现形式。

表 6-7　位运算符在 Kotlin 和 Java 中的展现形式对比

位运算说明	Kotlin 的位运算符	Java 的位运算符
按位与	and	&
按位或	or	|
按位异或	xor	^
按位左移	shl	<<
按位右移	shr	>>
无符号右移，高位补 0	ushr	>>>

6.3.2 图像视图 ImageView

图像视图是另一种常用的基本控件，在基本的图文控件之中，TextView 用于显示文本内容，而 ImageView 用于显示图像信息。

图像视图 ImageView 在代码中调用的方法说明如下。

- setImageDrawable：设置图形的 Drawable 对象。
- setImageResource：设置图形的资源 ID。
- setImageBitmap：设置图形的位图对象。
- setScaleType：设置图形的拉伸类型，在 Kotlin 中可直接给属性 scaleType 赋值。拉伸类型的取值说明见表 6-8。

表 6-8 拉伸类型的取值说明

ScaleType 类的拉伸类型	说明
ScaleType.FIT_XY	拉伸图片使之正好填满视图（图片可能被拉伸变形）
ScaleType.FIT_START	拉伸图片使之位于视图上部
ScaleType.FIT_CENTER	拉伸图片使之位于视图中间
ScaleType.FIT_END	拉伸图片使之位于视图下部
ScaleType.CENTER	保持图片原尺寸，并使之位于视图中间
ScaleType.CENTER_CROP	拉伸图片使之充满视图，并位于视图中间
ScaleType.CENTER_INSIDE	使图片位于视图中间（只压不拉）。当图片尺寸大于视图时，centerInside 等同于 fitCenter；当图片尺寸小于视图时，centerInside 等同于 center

读者应该注意到了，ImageView 的拉伸类型种类繁多，且文字说明不易理解，特别是与 center 相关的类型就有 4 种：fitCenter、center、centerCrop、centerInside，真是要把人搞晕了。接下来还是进行实验，把一张图片放入图像视图，然后尝试运用不同的拉伸类型，看看它们之间究竟有什么区别。下面是图片拉伸演示用到的 Kotlin 代码例子：

```
    class ImageScaleActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_image_scale)
            iv_scale.setImageResource(R.drawable.apple1)
            btn_center.setOnClickListener { iv_scale.scaleType = ScaleType.CENTER }
            btn_fitCenter.setOnClickListener { iv_scale.scaleType =
ScaleType.FIT_CENTER }
            btn_centerCrop.setOnClickListener { iv_scale.scaleType =
ScaleType.CENTER_CROP }
            btn_centerInside.setOnClickListener { iv_scale.scaleType =
ScaleType.CENTER_INSIDE }
```

```
            btn_fitXY.setOnClickListener { iv_scale.scaleType = ScaleType.FIT_XY }
            btn_fitStart.setOnClickListener { iv_scale.scaleType =
ScaleType.FIT_START }
            btn_fitEnd.setOnClickListener { iv_scale.scaleType =
ScaleType.FIT_END }
        }
    }
```

运行上述演示代码，可见图像拉伸的效果如图 6-20～图 6-23 所示。其中图 6-20 展示 fitCenter 效果，此时图片被拉伸但未超出控件范围；图 6-21 展示 center 效果，此时图片没有拉伸；图 6-22 展示 centerCrop 效果，此时图片被拉伸且已超出控件范围；图 6-23 展示 centerInside 效果，此时图片没有拉伸。

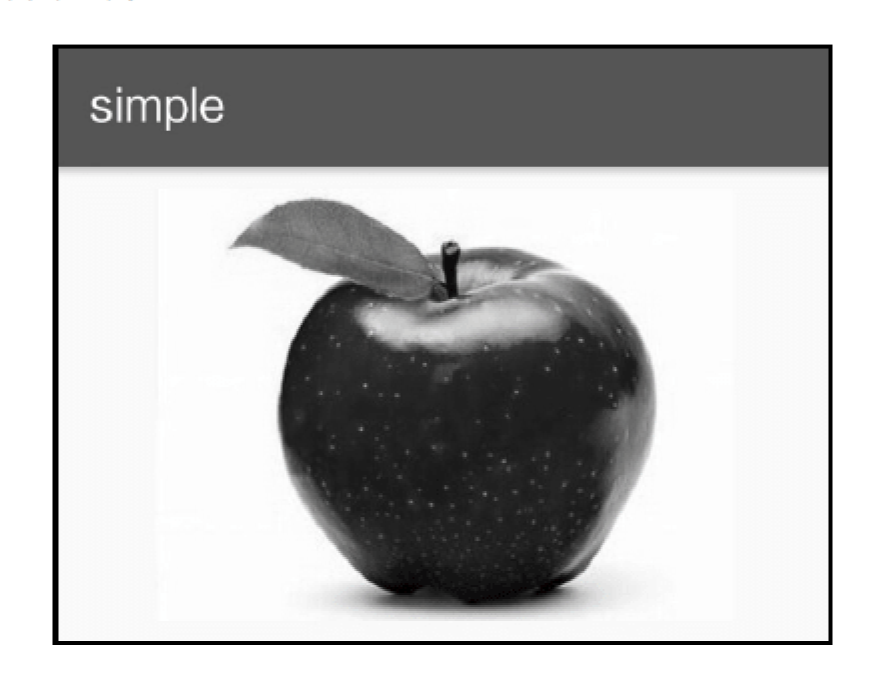


图 6-20　图片的 fitCenter 拉伸效果

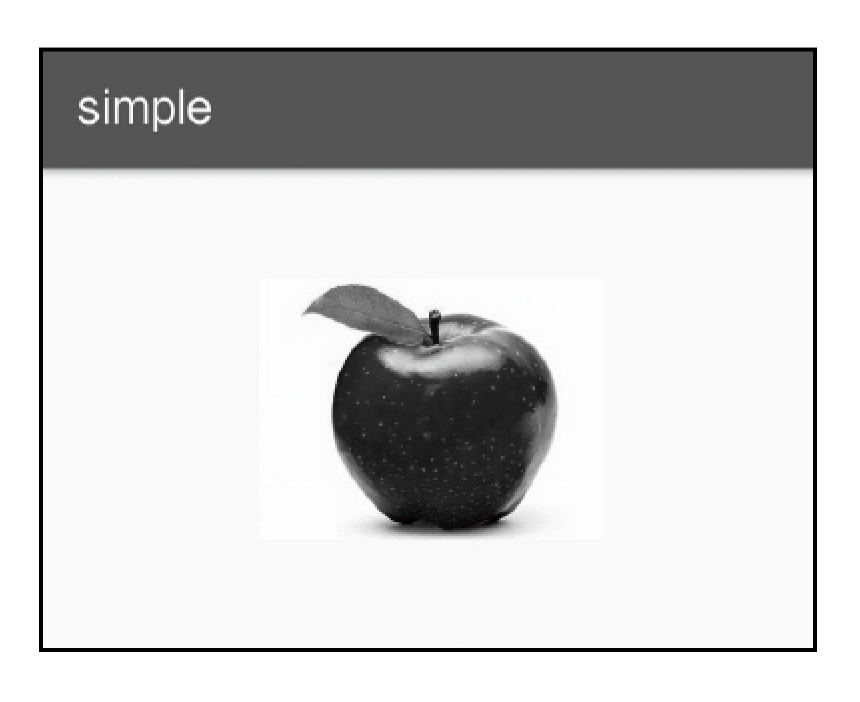


图 6-21　图片的 center 拉伸效果

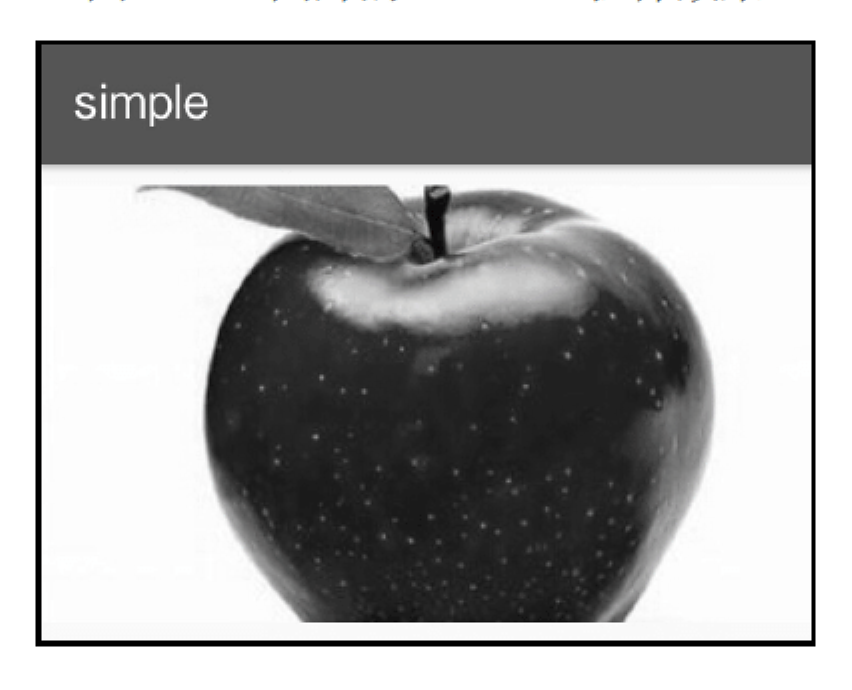


图 6-22　图片的 centerCrop 拉伸效果

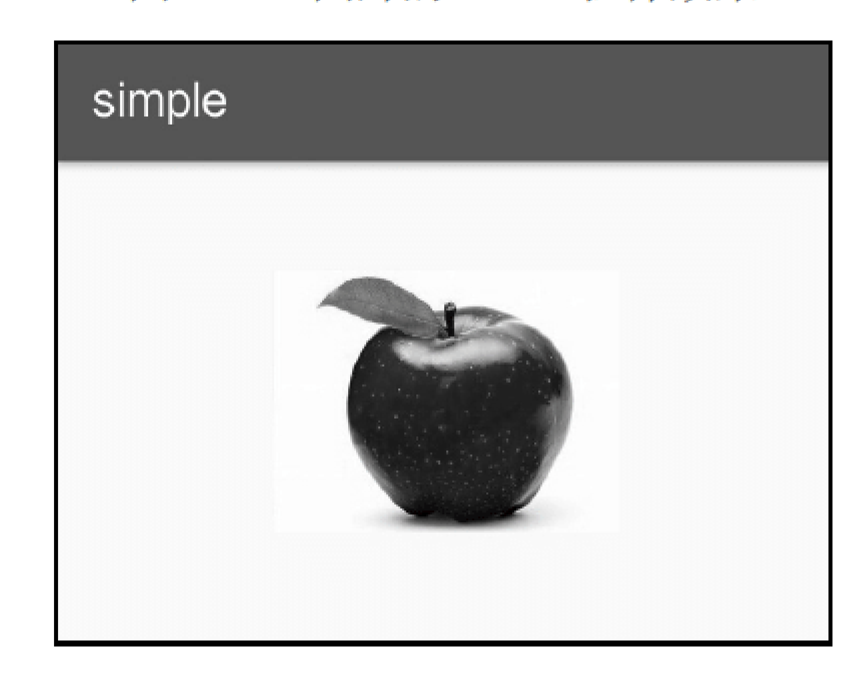


图 6-23　图片的 centerInside 拉伸效果

6.3.3　文本编辑框 EditText

前面介绍的文本视图只能显示事先设定好的文本，也就是说，TextView 能够输出文本但不能输入文本。要想监听用户输入的文本，就要用到文本编辑框 EditText，用户可在编辑框里面输入包括文字、数字、标点在内的文本信息。

为了规范用户的输入信息，EditText 提供了 setInputType 方法，用于过滤合法的输入字符，只有符合输入类型的字符，才允许接收并显示出来。Kotlin 可直接给 inputType 属性设置输入类型，从而取代 setInputType 的方法调用，常见的输入类型取值说明见表 6-9。

表 6-9 常见输入类型的取值说明

InputType 类的输入类型	说明
InputType.TYPE_CLASS_TEXT	所有文本
InputType.TYPE_CLASS_NUMBER	只能是数字
InputType.TYPE_CLASS_DATETIME	只能是日期时间
InputType.TYPE_TEXT_VARIATION_NORMAL	正常显示
InputType.TYPE_TEXT_VARIATION_PASSWORD	密文显示
InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD	明文密码

可是输入类型仅仅检验字符的有效性，尚缺少更灵活的加工处理。比如用户输入 11 位的手机号码的时候，能否在输完 11 位数字后自动触发某项动作？又比如有时希望输入的文本不包含回车符和换行符，能否自动屏蔽用户不小心输入的回车和换行符？为了解决以上问题，EditText 内置了一个编辑观察器 EditWatcher，它可以实时监控用户的输入字符，并且支持在输入每个字符时由开发者进行手工干预，从而实现随时校验、随时加工的功能。

下面是演示编辑观察器处理的 Kotlin 代码例子：

```
    class EditTextActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_edit_text)
            //注意不能直接给 EdiText 控件的 text 属性赋值
            //否则会报错 Editable 与 String 类型不匹配
            //只能调用 setText 方法对 EdiText 控件设置文本
            et_phone.setText("");
            //显示明文数字
            et_phone.inputType = InputType.TYPE_CLASS_NUMBER
            //显示明文密码
            et_phone.inputType = InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD
            //隐藏密码
            et_phone.inputType = InputType.TYPE_CLASS_TEXT or
InputType.TYPE_TEXT_VARIATION_PASSWORD
            //给编辑框添加文本变化的监听器
            et_phone.addTextChangedListener(EditWatcher())
        }

        private inner class EditWatcher : TextWatcher {

            override fun beforeTextChanged(s: CharSequence, start: Int, count: Int,
after: Int) {}

            override fun onTextChanged(s: CharSequence, start: Int, before: Int,
count: Int) {}

            override fun afterTextChanged(s: Editable) {
```

```
                var str = s.toString()
                //发现输入回车符或换行符
                if (str.indexOf("\r") >= 0 || str.indexOf("\n") >= 0) {
                    //去掉回车符和换行符
                    str = str.replace("\r", "").replace("\n", "")
                }
                if (str.length >= 11) {
                    tv_phone.text = "您输入的手机号码是：$str"
                }
            }
        }
    }
```

编辑观察器 TextWatcher 的运行结果如图 6-24 和图 6-25 所示，其中图 6-24 所示为输入 10 位手机号码时的界面，图 6-25 所示为输入第 11 位手机号码时的界面，可见输满 11 位手机号就会触发输入完成动作。



图 6-24　输入 10 位手机号码时的界面

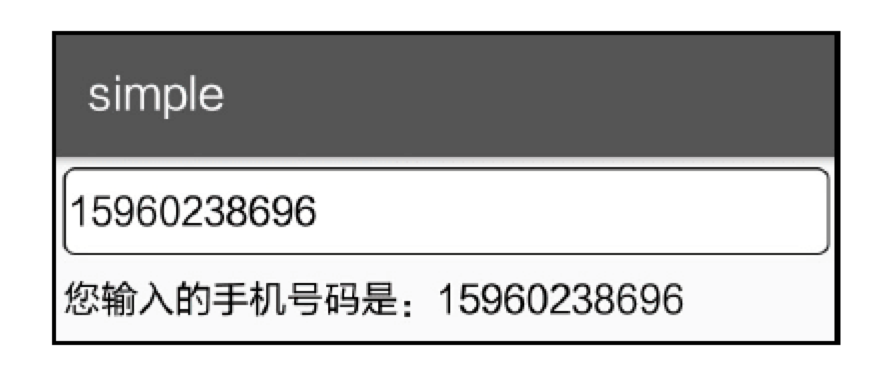


图 6-25　输入 11 位手机号码时的界面

6.4　Activity 活动跳转

活动 Activity 是 Android 最常用的组件，一般来说，一个 Activity 就代表一个页面，所以活动的用法就围绕着页面的运行过程而展开，包括活动的生命周期、活动是如何启动的、活动页面之间是如何传递信息的（包括发送数据和返回数据），等等。本节就对各种 Activity 用法对应的 Kotlin 实现方式进行逐步的阐述。

6.4.1　传送配对字段数据

Activity 的活动页面跳转是 App 常用的功能之一，在前几章的 DEMO 源码中便多次见到，常常是点击界面上的某个按钮，然后跳转到与之对应的下一个页面。对于 App 开发者来说，该功能的实现非常普通，使用 Java 编码不过以下两行代码而已：

```
    Intent intent = new Intent(MainActivity.this, LinearLayoutActivity.class);
    startActivity(intent);
```

上面代码的关键之处在于 Intent 的构造函数，其中第一个参数指定了页面跳转动作的来源，即 MainActivity 这个源页面，MainActivity.this 通常简写为 this；构造 Intent 的第二个参数则表示页面

跳转动作的目的地，即 LinearLayoutActivity 这个目标页面。倘若把这两行 Java 代码转换为 Kotlin 代码（复制这两行然后粘贴到 kt 文件中，Android Studio 就会自动完成转换），则可看到活动跳转的 Kotlin 代码如下所示：

```
val intent = Intent(this@MainActivity, LinearLayoutActivity::class.java)
startActivity(intent)
```

对比之下，这里的 Kotlin 代码与 Java 代码主要有两点不同之处：

（1）在类内部指代自身的 this 关键字，Java 的完整写法是“类名.this”，而 Kotlin 的完整写法是“this@类名”，当然二者均可简写为“this”。

（2）获取某个类的 class 对象，Java 的写法是“类名.class”，而 Kotlin 的写法是“类名::class.java”，一看便知带有浓浓的 Java 风味。

看起来，Kotlin 代码与 Java 代码半斤八两，未有明显的简化，令人产生小小的失望。但细心的读者也许已经注意到了，前几章附录源码里的活动跳转并非上述的 Kotlin 正宗写法，而是下面这种简化版的写法：

```
startActivity<LinearLayoutActivity>()
```

究其原因，乃是 Anko 库利用 Kotlin 的扩展函数给 Context 类新增了名为 startActivity 的新方法。故而使用简化版的写法之前，必须先导入 Anko 库的指定代码，即在 kt 文件头部添加下面一行导入语句：

```
import org.jetbrains.anko.startActivity
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

活动页面跳转的时候，往往还要携带一些请求参数，如果使用 Java 编码，可以很轻松地调用 Intent 对象的 putExtra 方法，通过“putExtra(参数名, 参数值)”的方式传递消息，就像下面的代码那样：

```
Intent intent = new Intent(this, ActSecondActivity.class);
intent.putExtra("request_time", DateUtil.getNowTime());
intent.putExtra("request_content", et_request.getText().toString());
startActivity(intent);
```

如果使用 Anko 的简化写法，其实也很容易，只要在 startActivity 后面的括号中依次填上每个参数字段的字段名和字段值，传送跳转参数的具体 Kotlin 代码如下所示：

```
//第一种写法，参数名和参数值使用关键字 to 隔开
startActivity<ActSecondActivity>(
      "request_time" to DateUtil.nowTime,
      "request_content" to et_request.text.toString())
```

注意到上面的写法使用关键字 to 隔开参数名和参数值，感觉不够美观，而且容易使人迷惑，to 后面究竟要跟着字段名还是字段值呢？所以 Anko 库提供了另一种符合习惯的写法，也就是利用

Pair 类把参数名和参数值进行配对，Pair 的第一个参数为字段名，第二个参数为字段值。据此改写后的 Kotlin 跳转代码如下所示：

```
        //第二种写法，利用 Pair 把参数名和参数值进行配对
        startActivity<ActSecondActivity>(
                Pair("request_time", DateUtil.nowTime),
                Pair("request_content", et_request.text.toString()))
```

无论哪种写法，在下一个活动中解析请求参数的方式都一样，都得先获取 Bundle 对象，然后分别根据字段名称获取对应的字段值。解析请求参数的具体 Kotlin 代码如下所示：

```
    class ActSecondActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_act_second)
            //获得请求参数的包裹
            val bundle = intent.extras
            val request_time = bundle.getString("request_time")
            val request_content = bundle.getString("request_content")
            tv_response.text = "收到请求消息：\n 请求时间为${request_time}\n 请求内容为
${request_content}"
        }
    }
```

下面通过测试界面观察一下消息数据发送之前和发送之后的效果，如图 6-26 所示，这时第一个界面准备跳转到第二个界面；如图 6-27 所示，这是跳转后的第二个界面，此时界面上展示第一个界面传递过来的参数信息。

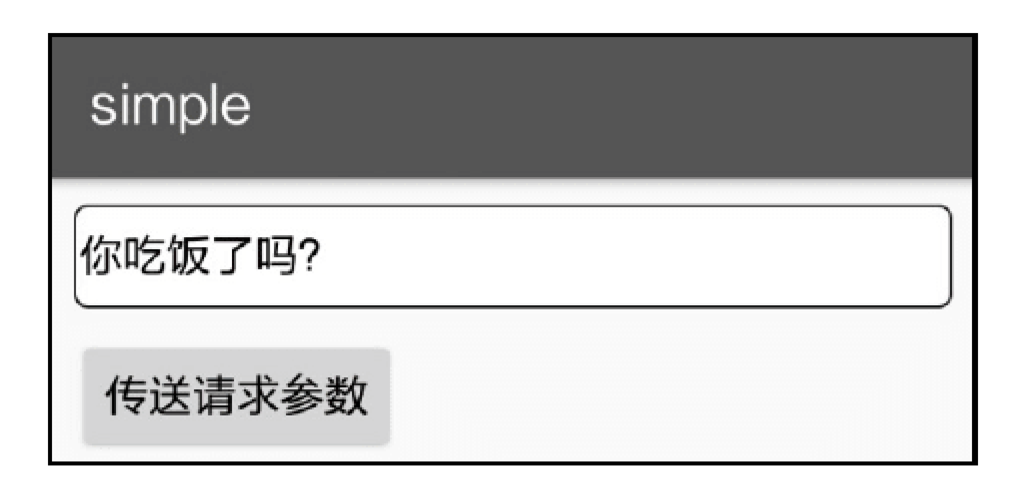


图 6-26　第一个界面准备跳转

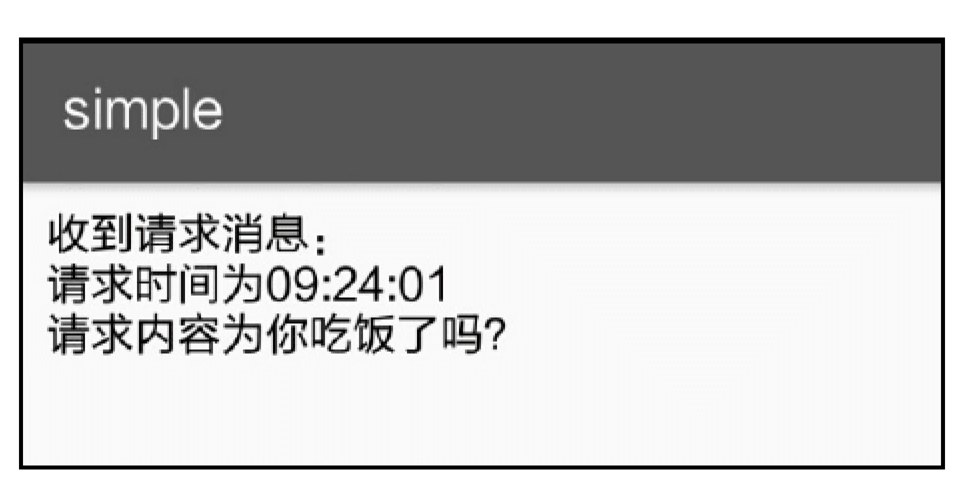


图 6-27　第二个界面接收请求

6.4.2　传送序列化数据

Activity 之间传递的参数类型除了整型、浮点型、字符串等基本数据类型外，还允许传递序列化结构，如 Parcelable 对象。这个 Parcelable 对象可不是简单的实体类，而是实现了 Parcelable 接口的实体类，实现接口意味着该类必须重写接口定义的所有方法，无论你愿不愿意都得老老实实地依样画葫芦。譬如，前面的活动跳转传递了两个字段数据，如果把这两个字段放到 Parcelable 对象中，仅仅包含两个字段的 Parcelable 类对应的 Java 代码也如下面这般冗长：

```
public class MessageInfo implements Parcelable {
    public String content;
    public String send_time;

    // 写数据
    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeString(content);
        out.writeString(send_time);
    }

    // 例行公事实现 createFromParcel 和 newArray
    public static final Parcelable.Creator<MessageInfo> CREATOR
            = new Parcelable.Creator<MessageInfo>() {
        // 读数据
        public MessageInfo createFromParcel(Parcel in) {
            MessageInfo info = new MessageInfo();
            info.content = in.readString();
            info.send_time = in.readString();
            return info;
        }

        public MessageInfo[] newArray(int size) {
            return new MessageInfo[size];
        }
    };

    @Override
    public int describeContents() {
        return 0;
    }
}
```

看看这架势，如此简单的自定义 Parcelable 序列化结构类，就得重写包括 writeToParcel、createFromParcel、newArray、describeContents 在内的 4 个方法，可谓是兴师动众。由此可见，这里又是 Java 的一个痛点，正适合 Kotlin 施展拳脚、好好改进。在第 5 章的类和对象中介绍了 Kotlin 对数据类的写法，只要在类名前面加入关键字 data，Kotlin 即可自动提供 get/set、equals、copy、toString 等诸多方法。那么序列化对象的改造也相当简单，仅需在类名之前增加一行注解“@Parcelize”就好了，于是整个类的 Kotlin 代码只有下面寥寥几行：

```
//@Parcelize 注解表示自动实现 Parcelable 接口的相关方法
@Parcelize
data class MessageInfo(val content: String, val send_time: String) :
Parcelable {
}
```

不过若想正常编译，还需修改模块的编译文件 build.gradle，在文件末尾添加下面几行，表示增加对安卓插件的编译支持：

```
//@Parcelize 标记需要设置 experimental = true
androidExtensions {
    experimental = true
}
```

编译文件修改完毕，现在能在 Kotlin 中使用序列化对象的注解了。虽然自定义的 MessageInfo 类内部没有任何一行代码，但是它除了具备数据类的所有方法外，也自动实现了 Parcelable 接口的几个方法。接下来就可以利用该类传输活动跳转的序列化数据了，下面是改写后的 Kotlin 跳转代码：

```
    val request = MessageInfo(et_request.text.toString(), DateUtil.nowTime)
    startActivity<ParcelableSecondActivity>("message" to request)
```

跳转后的下一个页面调用 getParcelable 即可正常获得原始的序列化数据，具体的数据解析 Kotlin 代码如下所示：

```
class ParcelableSecondActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_parcelable_second)
        //获得 Parcelable 格式的请求参数
        val request = intent.extras.getParcelable<MessageInfo>("message")
        tv_response.text = "收到打包好的请求消息：\n 请求时间为
${request.send_time}\n 请求内容为${request.content}"
    }
}
```

同样通过测试界面观察序列化对象的打包和解包效果，如图 6-28 所示，这时第一个页面准备携带序列化数据跳转到第二个页面；如图 6-29 所示，这是跳转后的第二个页面，此时界面上展示第一个页面传递过来的序列化数据。

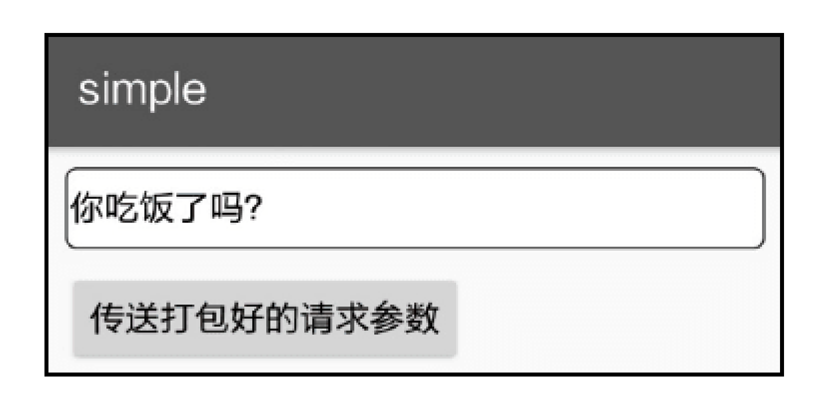


图 6-28　第一个页面传送序列化数据

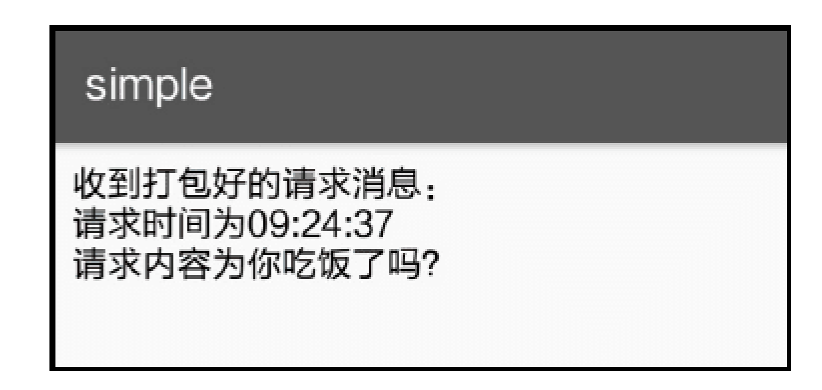


图 6-29　第二个页面接收序列化数据

6.4.3　跳转时指定启动模式

前面提到 Anko 库的 startActivity 方法取消了 Intent 意图对象，这个做法有利有弊，弊端是 Intent 对象还有其他的设置方法，比如 setAction 方法用来设置动作、setData 方法用来设置路径、

addCategory 方法用来设置动作类别、setType 方法用来设置数据类型、setFlags 方法用来设置启动模式等。所以对于复杂一点的活动跳转，必须要保留 Intent 对象，为此 Anko 库额外提供了 intentFor 方法，用于简单生成 Intent 对象，其书写格式类似于 startActivity 方法，举个例子：

```
val intent = intentFor<ActSecondActivity>(
      "request_time" to DateUtil.nowTime,
      "request_content" to et_request.text.toString())
```

这下通过 intentFor 方法得到了 Intent 对象，再去设置其他参数就方便了，活动跳转时也只需调用“startActivity(intent)”即可。

intentFor 方法的作用并不仅限于得到一个意图对象，还可以用来设置新活动的启动模式。启动模式是 Android 用来控制 Activity 活动栈行为的一种标志，在 App 运行期间，打开的页面会被逐个加入一个活动栈。一般来说，位于栈顶的是 App 首页，其后打开的页面依次加到栈尾，返回时从栈尾依次出栈。但是出于效率考虑，有时我们希望对栈的操作能够不按顺序处理，所以也就有了活动页面的启动模式。

Android 有两种方式用来设置 Activity 的启动模式，其一是修改 AndroidManifest.xml，在指定的 activity 节点添加属性 launchMode，表示本活动以该启动模式来运行；其二是在代码中给 Intent 对象调用 setFlags 方法，从而表明接下来打开的页面运用该启动标志。下面分别详细说明启动模式的两种设置方法。

1. 在配置文件中指定启动模式

第一种设置办法是在 AndroidManifest.xml 里的 activity 节点添加属性 launchMode，具体的 activity 节点配置内容示例如下：

```
<activity android:name=".ActSecondActivity" android:launchMode=
"standard" />
```

其中，launchMode 属性的几种取值说明见表 6-10。

表 6-10　launchMode 属性的取值说明

launchMode 属性值	说明
standard	标准模式，无论何时启动哪个 activity，都是重新创建该页面的实例并放入栈尾。如果不指定 launchMode 属性，就默认为标准模式
singleTop	启动 activity 时，判断栈顶正好是该 Activity 的实例，就重用该实例；否则创建新的实例并放入栈顶，否则的情况与 standard 类似
singleTask	启动 activity 时，判断栈中存在该 Activity 的实例，就重用该实例，并清除位于该实例上面的所有实例；否则的情况处理同 standard
singleInstance	启动 activity 时，将该 Activity 的实例放入一个新栈中，原栈的实例列表保持不变

2. 在代码里面设置启动标志

第二种设置办法是在代码中给 Intent 对象调用 setFlags 方法，具体的方法调用代码如下　所示：

```
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

之所以要在代码中动态指定活动页面的启动模式，是因为 AndroidManifest.xml 中对每个 Activity 只能指定唯一的启动模式。如果想在不同时候对同一个 Activity 运用不同的启动模式，显然固定的 launchMode 属性无法满足这个要求。于是 Android 允许在代码中手动设置启动标志，这样在不同时候调用 startActivity 方法就能运行特定的启动模式。

适用于 setFlags 方法的几种启动标志的取值说明见表 6-11。

表 6-11 代码中的启动标志取值说明

Intent 类的启动标志	说明
Intent.FLAG_ACTIVITY_NEW_TASK	开启一个新任务，flags 默认该值。该值类似于 launchMode="standard"，不同之处在于，如果原来不存在活动栈，FLAG_ACTIVITY_NEW_TASK 就会创建一个新栈
Intent.FLAG_ACTIVITY_SINGLE_TOP	当栈顶为待跳转的 activity 实例时，重用栈顶的实例。该值等同于 launchMode="singleTop"
Intent.FLAG_ACTIVITY_CLEAR_TOP	当栈中存在待跳转的 activity 实例时，重新创建一个新实例，并将原实例上方的所有实例加以清除。该值与 launchMode="singleTask"类似，但 launchMode="singleTask"采用 onNewIntent 启用原任务，而 FLAG_ACTIVITY_CLEAR_TOP 采用先 onDestroy 再 onCreate 创建新任务
Intent.FLAG_ACTIVITY_NO_HISTORY	该标志与 launchMode="standard"情况类似，但栈中不保存新启动的 activity 实例。这样下次无论以何种方式再启动该实例，也要走 standard 的完整流程
Intent.FLAG_ACTIVITY_CLEAR_TASK	该标志非常暴力，跳转到新页面时，栈中的原有实例都被清空。注意，该标志需要结合 FLAG_ACTIVITY_NEW_TASK 使用，即 setFlags 的参数为“Intent.FLAG_ACTIVITY_CLEAR_TASK|Intent.FLAG_ACTIVITY_NEW_TASK”

讲了这么多的启动模式，可见其概念很是琐碎，并且设置启动标志的代码也较啰嗦，所以 Kotlin 利用 Anko 库扩展出来的 intentFor 函数简化启动标志的设置方式。例如，启动标志 FLAG_ACTIVITY_NEW_TASK 对应的 Anko 写法，连同页面跳转动作也只需下面短短的一行 Kotlin 代码：

```
startActivity(intent.newTask())
```

优化后的启动标志设置函数 newTask 延续了 Kotlin 一贯的简洁风格，至于其他的启动标志，则可将“newTask”替换为对应的设置方法。Java 的启动标志与 Anko 的标志设置函数对应关系见表 6-12。

表 6-12 启动标志的 Anko 方法与 Java 的对应关系

Intent 类的启动标志	Anko 库的标志设置函数
Intent.FLAG_ACTIVITY_NEW_TASK	newTask()
Intent.FLAG_ACTIVITY_SINGLE_TOP	singleTop()

（续表）

Intent 类的启动标志	Anko 库的标志设置函数
Intent.FLAG_ACTIVITY_CLEAR_TOP	clearTop()
Intent.FLAG_ACTIVITY_NO_HISTORY	noHistory()
Intent.FLAG_ACTIVITY_CLEAR_TASK	clearTask()

6.4.4　处理返回数据

页面跳转的多数情况是上一个页面传递请求参数给下一个页面，当然也有少数情况是上一个页面需要接收下一个页面的返回数据。此时 Kotlin 跟 Java 一样都采取 startActivityForResult 方法，表示这次活动跳转要求处理返回信息，具体的 Kotlin 代码举例如下：

```
            val info = MessageInfo(et_request.text.toString(), DateUtil.nowTime)
            //ForResult 表示需要返回参数
            startActivityForResult<ActResponseActivity>(0, "message" to info)
```

那么下一个页面返回应答参数的 Kotlin 代码也跟 Java 的做法类似，一样要调用 setResult 方法将应答参数返回给上一个页面：

```
        btn_act_response.setOnClickListener {
            val response = MessageInfo(et_response.text.toString(),
DateUtil.nowTime)
            val intent = Intent()
            intent.putExtra("message", response)
            //调用 setResult 方法表示携带应答参数返回到上一个页面
            setResult(Activity.RESULT_OK, intent)
            finish()
        }
```

上一个页面接收到了返回的应答数据，将它解包获得原始的字段信息，再做进一步的处理。下面是上一个页面处理返回数据的 Kotlin 代码例子：

```
        //从下一个页面返回到本页面时回调 onActivityResult 方法
        override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
            if (data != null) {
                //获得下一个页面的应答参数
                val response = data.extras.getParcelable<MessageInfo>("message")
                tv_request.text = "收到返回消息：\n 应答时间为${response.send_time}\n
应答内容为${response.content}"
            }
        }
```

以上页面跳转与返回的数据处理结果如图 6-30 和图 6-31 所示，其中图 6-30 所示为跳到下一个页面时的界面截图，图 6-31 所示为返回到上一个页面时的界面截图。

simple
收到打包好的请求消息：
请求时间为09:25:01
请求内容为你吃饭了吗?
吃过了。你呢?
返回应答参数

图 6-30　下一个页面准备返回应答消息

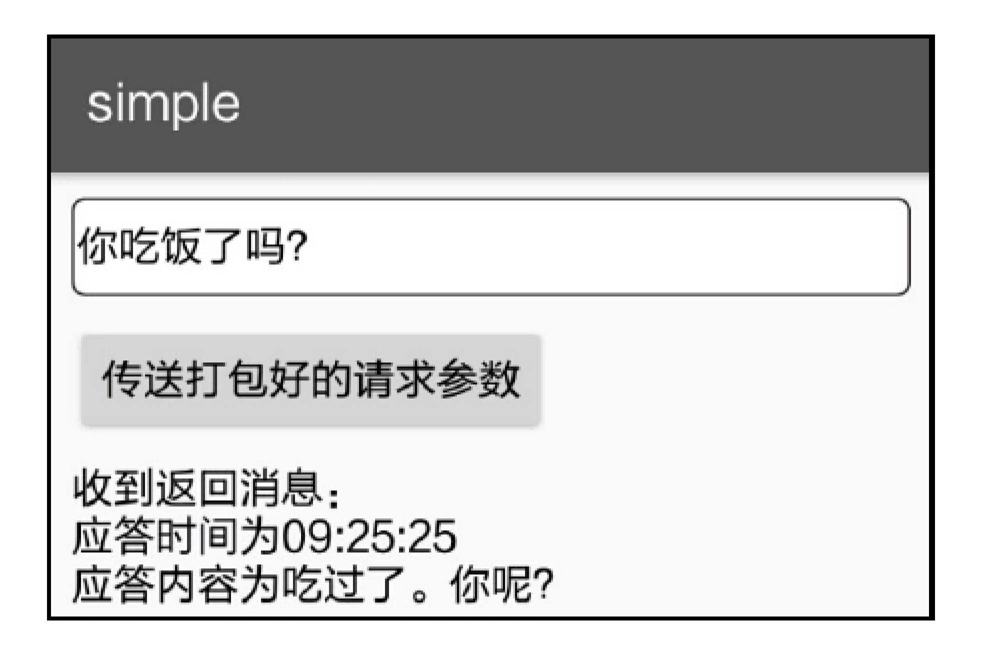


图 6-31　上一个页面接收返回的消息

6.5　实战项目：电商 App 的登录页面

在这个移动互联网时代，用户是每家 IT 企业最宝贵的资源，但凡一个数得上号的 App，无不坐拥几千万乃至数亿的用户，只有掌握了足够多的用户，开发 App 的公司才会被资本青睐，才有可能在激烈的市场竞争中生存下去。那么对于用户来说，首次打开一个电商 App 又会做什么事呢？浏览商品固然是必做的功课，但对于 App 而言，吸引用户注册并登录才是紧要之事，因为用户登录之后才有机会产生商品交易。于是登录功能必然是电商 App 的一个重要模块，本节的实战项目就来谈谈如何利用 Kotlin 开发一个功能完备的登录页面。

6.5.1　需求描述

各家电商 App 的登录页面大同小异，要么是用户名与密码组合登录，要么是手机号与验证码组合登录，若是做好一点的，则会提供忘记密码与记住密码等功能。当然做需求只有文字描述肯定是不行的，一定要有界面效果图配合讲解才可以。下面先来看看登录页面的效果图，因为有两种组合登录方式，所以登录页面也分成两个效果图。如图 6-32 所示，这是选中密码登录方式时的界面；如图 6-33 所示，这是选中验证码登录时的界面。

从以上两个登录效果图看到，密码登录与验证码登录的界面主要存在以下几点区别：

（1）密码/验证码输入框的左侧标题以及输入框内部的提示语各不相同。

（2）如果是密码登录，就需要支持找回密码；如果是验证码登录，就需要支持向用户手机发送验证码。

（3）密码登录可以提供记住密码功能，而验证码的数值每次都不一样，无须记住，也没法记住。

对于找回密码功能，一般是跳到单独的找回密码页面，在该页面输入和确认新密码，并校验找回密码的合法性（通过短信验证码检查），据此勾勒出密码找回页面的轮廓概貌，如图 6-34 所示。

在找回密码的操作过程当中，为了更好地增强用户体验，有必要在几个关键节点处提醒用户。比如成功发送验证码之后，要及时提示用户注意查收短信，这里暂且做成提示对话框的形式，如图 6-35 所示。又比如密码登录成功之后，也要告知用户已经成功登录，注意继续后面的操作，登录成功的提示信息如图 6-36 所示。



图 6-32　选中密码登录方式时的界面



图 6-33　选中验证码登录时的界面



图 6-34　找回密码页面的界面效果

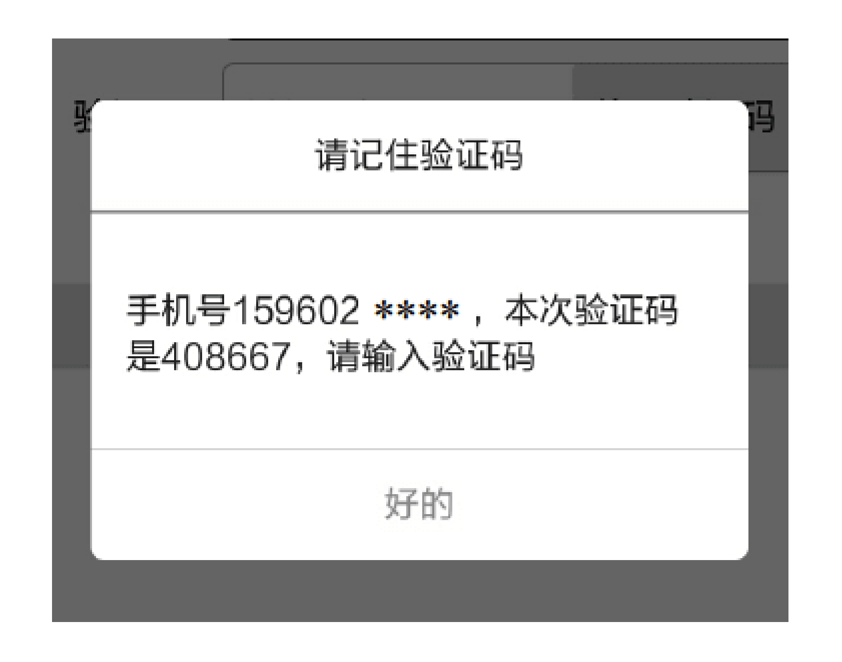


图 6-35　发送验证码的提示框

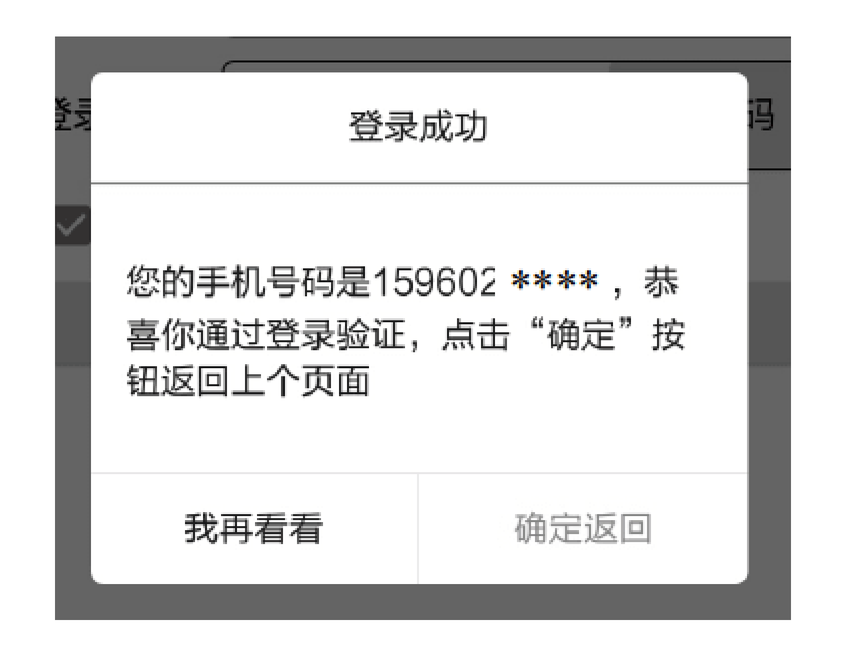


图 6-36　登录成功后的提示框

真是想不到，原来简简单单的一个登录功能就得考虑这么多需求场景。可是仔细想想，这些需求场景都是必要的，其目的是为了让用户能够更加便捷地顺利登录。正所谓“台上十分钟，台下十年功”，每个好用的 App 都离不开程序员数十年如一日的辛勤工作。

6.5.2　开始热身：提醒对话框 AlertDialog

手机上的 App 极大地方便了人们的生活，很多业务只需用户拇指一点即可轻松办理，然而这也带来了一定的风险，因为有时候用户并非真的想这么做，只是不小心点了一下而已，如果 App 不做任何提示的话，继续吭哧吭哧兀自办完业务，比如转错钱了、误删资料了、密码输错了，造成不可挽回的后果往往令用户追悔莫及。所以对于部分关键业务，App 为了避免用户的误操作，很有必要弹出消息对话框，提醒用户是否真的要进行此项操作。

这个提醒对话框便是 App 开发常见的 AlertDialog，说起这个 AlertDialog，安卓开发者都有所耳闻，该对话框不外乎消息标题、消息内容、确定按钮、取消按钮这 4 个要素，若是使用 Java 编码显示提醒对话框，基本跟下面的示例代码大同小异：

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("尊敬的用户");
builder.setMessage("你真的要卸载我吗？");
builder.setPositiveButton("残忍卸载", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        tv_alert.setText("虽然依依不舍，但是只能离开了");
    }
```

```
    });
    builder.setNegativeButton("我再想想", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            tv_alert.setText("让我再陪你三百六十五个日夜");
        }
    });
    AlertDialog alert = builder.create();
    alert.show();
```

显而易见，上述的 Java 代码非常冗长，特别是两个按钮的点击事件，既有匿名类又有函数重载，令人不堪卒读。现在尝试将以上的 Java 代码转换为 Kotlin 代码，转换后的 Kotlin 代码如下所示：

```
    val builder = AlertDialog.Builder(this)
    builder.setTitle("尊敬的用户")
    builder.setMessage("你真的要卸载我吗？")
    builder.setPositiveButton("残忍卸载") { dialog, which -> tv_alert.text = "虽然依依不舍，还是只能离开了" }
    builder.setNegativeButton("我再想想") { dialog, which -> tv_alert.text = "让我再陪你三百六十五个日夜" }
    val alert = builder.create()
    alert.show()
```

这下看来点击事件的代码在很大程度上简化了，不过除此之外，整块代码依然显得有些臃肿，尤其是运用了建造者模式的 Builder 类，虽然表面上增强了安全性，但对于编码来说其实是累赘。因此，Anko 库将其做了进一步的封装，其实就是给 Context 类添加了一个扩展函数 alert，具体格式如“alert(消息内容, 消息标题) { 几个按钮及其点击事件 }”，简化后的 Kotlin 弹窗代码如下所示：

```
    alert("你真的要卸载我吗？", "尊敬的用户") {
        positiveButton("残忍卸载") { tv_alert.text = "虽然依依不舍，还是只能离开了" }
        negativeButton("我再想想") { tv_alert.text = "让我再陪你三百六十五个日夜" }
    }.show()
```

现在的 Kotlin 代码相比之下更方便阅读了，并且代码量还不到原来 Java 代码的三分之一。当然，为了正常使用这么好的扩展函数，不要忘了在代码文件头部加入下面一行导入语句：

```
    import org.jetbrains.anko.alert
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
        compile "org.jetbrains.anko:anko-common:$anko_version"
```

这么精简的 Kotlin 代码，功能上可是一点都没有偷工减料，它的提醒对话框效果与 Java 编码一模一样，都如图 6-37 所示。

点击提醒对话框两个按钮之后的界面分别如图 6-38 和图 6-39 所示，其中图 6-38 所示为点击“残忍卸载”之后的界面，图 6-39 所示为点击“我再想想”之后的界面。



图 6-37　提醒对话框的例子效果

simple
弹出提醒对话框
虽然依依不舍，但是只能离开了

图 6-38　点击“残忍卸载”之后的界面

simple
弹出提醒对话框
让我再陪你三百六十五个日夜

图 6-39　点击“我再想想”之后的界面

6.5.3　控件设计

用户登录界面看似简单，用到的控件却不少。按照之前登录界面的效果图，大致从上到下、从左到右分布着下列 Android 控件。

- 单选按钮 RadioButton：用来区分是密码登录还是验证码登录。
- 文本视图 TextView：输入框左侧要显示此处应该输入什么信息。
- 编辑框 EditText：用来输入手机号码和密码。
- 复选框 CheckBox：用于判断是否记住密码。
- 按钮 Button：除了“登录”按钮之外，还有“忘记密码”和“获取验证码”两个按钮。
- 线性布局 LinearLayout：指定手机号码的文本视图与手机号码的编辑框从左到右依次排列。
- 相对布局 RelativeLayout：忘记密码的按钮与密码输入框是叠加的，且“忘记密码”与上级视图右对齐。
- 单选组 RadioGroup：密码登录和验证码登录这两个单选按钮需要放在单选组布局之中。

另外，由于整个登录模块由登录页面和找回密码页面组成，因此这两个页面之间需要进行数据交互，也就是在页面跳转时传递参数。譬如，从登录页面跳到找回密码页面要携带唯一标识的手机号码作为请求参数，不然密码找回页面不知道要给哪个手机号修改密码。同时，从找回密码页面回到登录页面，也要将修改之后的新密码作为应答参数传回去，否则登录页面不知道密码被改成什么了。

6.5.4　关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成登录页面项目，下面列举几个重要功能的 Kotlin 代码片段。

1. 关于自动清空错误的密码

这里有个细微的用户体验问题：用户会去找回密码，肯定是发现输入的密码不对，那么修改密码完回到登录页面，密码输入框里还是刚才的错误密码，此时用户只能先清空错误密码，然后才能输入新密码。一个 App 要想让用户觉得好用，就得急用户之所急，想用户之所想，像刚才那个错误密码的情况，应当由 App 在返回登录页面时自动清空原来的错误密码。

自动清空密码框的操作放在 onActivityResult 方法中处理是个办法，但这样有个问题，如果用户直接按返回键回到登录页面，那么 onActivityResult 方法发现数据为空便不做处理。因此应该这么处理：判断当前是否为返回页面动作，只要是从找回密码页面返回到当前页面，无论是否携带应

答参数，均需自动清空密码输入框。对应的 Kotlin 代码则为重写登录页面的 onRestart 方法，在该方法中强制清空密码。这样一来，无论用户是修改密码完成回到登录页，还是点击返回键回到登录页，App 都会自动清空密码框。

下面是重写 onRestart 函数之后的 Kotlin 代码示例：

```
//从修改密码页面返回登录页面，要清空密码的输入框
override fun onRestart() {
    et_password.setText("")
    super.onRestart()
}
```

2. 关于自动隐藏输入法面板

在输入手机号码或者密码的时候，屏幕下方都会弹出输入法面板，供用户按键输入数字和字母。但是输入法面板往往占据屏幕下方大块空间，很是碍手碍脚，用户输完 11 位的手机号码时，还得再按一下返回键来关闭输入法面板，接着才能继续输入密码。这里按返回键纯属庸人自扰，理想的做法是：一旦用户输完 11 位手机号码，App 就要自动隐藏输入法。同理，一旦用户输完 6 位密码或者 6 位验证码，App 也要自动隐藏输入法。要想让 App 具备这种智能的判断功能，就需给文本编辑框添加监听器，只要当前编辑框输入文本长度达到 11 位或者 6 位，App 就自动将输入法面板隐藏。

下面是实现智能隐藏功能的 Kotlin 监听器代码例子：

```
private inner class HideTextWatcher(private val mView: EditText) : TextWatcher {
    private val mMaxLength: Int = ViewUtil.getMaxLength(mView)
    private var mStr: CharSequence? = null

    override fun beforeTextChanged(s: CharSequence, start: Int, count: Int,
after: Int) {}

    override fun onTextChanged(s: CharSequence, start: Int, before: Int, count:
Int) {
        mStr = s
    }

    override fun afterTextChanged(s: Editable) {
        if (mStr.isNullOrEmpty())
            return
        if (mStr!!.length == 11 && mMaxLength == 11 || mStr!!.length == 6 &&
mMaxLength == 6) {
            //隐藏输入法面板，ViewUtil 类参见本书下载资源中的源代码
            ViewUtil.hideOneInputMethod(this@LoginPageActivity, mView)
        }
    }
}
```

3. 关于找回密码的跳转事件

因为“找回密码”与“获取验证码”共用一个按钮，所以点击事件需要区分当前是要找回密码还是获取验证码。如果是找回密码，就携带手机号码跳到密码找回页面；如果是获取验证码，就在界面上提示用户注意接收验证码。

下面是“找回密码”/“获取验证码”两个按钮事件合并后的 Kotlin 代码示例：

```
    private fun doForget() {
        val phone = et_phone.text.toString()
        if (phone.isBlank() || phone.length < 11) {
            toast("请输入正确的手机号")
            return
        }
        if (rb_password.isChecked) {
            //携带手机号码跳到密码找回页面
            startActivityForResult<LoginForgetActivity>(mRequestCode, "phone" to
phone)
        } else if (rb_verifycode.isChecked) {
            mVerifyCode = String.format("%06d", (Math.random() * 1000000 %
1000000).toInt())
            alert("手机号$phone，本次验证码是$mVerifyCode，请输入验证码", "请记住验证码") {
                positiveButton("好的") {  }
            }.show()
        }
    }
```

密码修改完毕，在登录页面获取新密码的 Kotlin 代码如下所示：

```
    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        if (requestCode == mRequestCode && data != null) {
            //用户密码已改为新密码
            mPassword = data.getStringExtra("new_password")
        }
    }
```

4. 关于密码修改的校验操作

由于密码对于用户来说是很重要的信息，因此必须认真校验新密码的合法性，务必做到万无一失才行。具体的密码修改校验可分为下列 4 个步骤：

（1）新密码和确认输入的新密码都要是 6 位数字。

（2）新密码和确认输入的新密码必须保持一致。

（3）用户输入的验证码必须和系统下发的验证码一致。

（4）密码修改成功之后，携带修改后的新密码返回登录页面。

根据以上的校验步骤，对应书写的 Kotlin 代码例子如下所示：

```
private fun doConfirm() {
    val password_first = et_password_first.text.toString()
    val password_second = et_password_second.text.toString()
    if (password_first.isBlank() || password_first.length < 6 ||
            password_second.isBlank() || password_second.length < 6) {
        toast("请输入正确的新密码")
        return
    } else if (password_first != password_second) {
        toast("两次输入的新密码不一致")
        return
    } else if (et_verifycode.text.toString() != mVerifyCode) {
        toast("请输入正确的验证码")
        return
    } else {
        toast("密码修改成功")
        //携带修改后的新密码返回登录页面
        val intent = Intent().putExtra("new_password", password_first)
        setResult(Activity.RESULT_OK, intent)
        finish()
    }
}
```

6.6 小　结

本章主要介绍了 Kotlin 如何实现几种简单控件的调用，包括常见的按钮控件（文本按钮、复选框、单选按钮）、主要的布局视图（线性布局、相对布局、约束布局）、基本的图文控件（文本视图、图像视图、文本编辑框）以及 Activity 活动组件的控制操作（携带参数、启动模式、返回参数）。最后设计了一个实战项目“电商 App 的登录页面”，在该项目的 Kotlin 编码中采用了前面介绍的大部分布局和控件，以及 Activity 跳转与返回时的消息请求与应答，另外还介绍了 Kotlin 对提醒对话框的用法。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会使用 Kotlin 操作常见的按钮控件，除了复习匿名函数、内部类、接口、字符串模板、分支语句等基础语法知识之外，还需掌握 Kotlin 的类型转换办法。

（2）学会使用 Kotlin 操作主要的布局视图，除了掌握这些布局的自身功能实现之外，还需掌握 Anko 库的像素转换方法、相对视图的相对位置设置方法。

（3）学会使用 Kotlin 操作基本的图文控件，除了掌握图文信息的各种特效处理之外，还需重点掌握 Kotlin 位运算符的概念及其用法。

（4）学会使用 Kotlin 操作 Activity 组件的跳转过程，包括携带配对参数跳转、携带序列化参数跳转、跳转时指定启动标志、携带应答参数返回上个页面等。

（5）学会使用 Anko 库简化提醒对话框的调用代码。

第 7 章

Kotlin 操纵复杂控件

第 6 章介绍的简单控件只是 Android 界面开发的基础，若想让 App 界面真正活泼起来，变得熠熠生辉，还需要各式各样的复杂控件来组合运用。这些复杂控件既有常见的视图排列控件，又有来自 MaterialDesign 库的新颖布局，更有横向滑动的页面切换控件。本章就从这些常用的复杂控件着手，一边介绍它们的常规用法，一边介绍如何使用 Kotlin 更好、更方便地操作这些控件。

7.1　使用视图排列

虽然 Android 提供了很多控件，但是这些控件不能随意堆砌，因为任何事物都要讲究章法，有章可循才不会乱套，所以 Android 还提供了遵循某种展现规则的视图排列控件，包括下拉框 Spinner、列表视图 ListView、网格视图 GridView、循环视图 RecyclerView 等。本节就对这几种视图排列控件进行详细的说明，特别阐述如何使用 Kotlin 实现它们各自的适配器编码。

7.1.1　下拉框 Spinner

对于某些固定值的条件选择，比如红、绿、蓝三原色选择其一，一月份到十二月份选择其中一个月份等，这些情况在 Android 中用到了下拉框 Spinner。界面上的 Spinner 控件一开始是一个右侧带向下箭头的文本，点击该文本会弹出一个选择对话框，选中某一项之后，对话框消失，同时界面上的文本替换为刚才选中的文本内容。只看下拉框的功能其实挺简单的，可是如果用 Java 代码实现，就得费一番功夫了。

下面便是调用 Spinner 控件的 Java 代码例子：

```
    private void initSpinner() {
        ArrayAdapter<String> starAdapter = new ArrayAdapter<String>(this,
                R.layout.item_select, starArray);
```

```
            starAdapter.setDropDownViewResource(R.layout.item_dropdown);
            Spinner sp = (Spinner) findViewById(R.id.sp_dialog);
            sp.setPrompt("请选择行星");
            sp.setAdapter(starAdapter);
            sp.setSelection(0);
            sp.setOnItemSelectedListener(new MySelectedListener());
        }

        private String[] starArray = {"水星", "金星", "地球", "火星", "木星", "土星"};
        class MySelectedListener implements OnItemSelectedListener {
            public void onItemSelected(AdapterView<?> arg0, View arg1, int arg2,
long arg3) {
                Toast.makeText(SpinnerDialogActivity.this, "你选择的行星是
"+starArray[arg2], Toast.LENGTH_LONG).show();
            }

            public void onNothingSelected(AdapterView<?> arg0) {}
        }
```

不出所料，这里再次体现了 Java 编码的尾大不掉，简简单单的功能在 Java 代码中被分解为以下几个专门的处理：

（1）定义一个数组适配器 ArrayAdapter，指定待选择的字符串数组以及每项文本的布局文件。

（2）定义一个选择监听器 OnItemSelectedListener，它在用户选中某项时触发，并响应文本项的选中事件。

（3）Spinner 控件依次设置选择对话框的标题、数组适配器、选择监听器、默认选项等。

我的天，这也太专业了吧，在产品经理看来，这只是个下拉框而已，有必要搞这么复杂吗？然而 Java 代码就是这么错综复杂，要想开发 Android，只能这么做，不然还有更好的法子吗？不信换成 Kotlin 试试？说时迟那时快，在 Android Studio 上面把 Spinner 上述的 Java 代码转换为 Kotlin，不一会儿就生成了如下的 Kotlin 代码：

```
        private fun initSpinner() {
            val starAdapter = ArrayAdapter(this, R.layout.item_select, starArray)
            starAdapter.setDropDownViewResource(R.layout.item_dropdown)
            //Android 8.0 之后的 findViewById 方法要求后面添加"<View>"才能进行类型转换操作
            val sp = findViewById<View>(R.id.sp_dialog) as Spinner
            sp.prompt = "请选择行星"
            sp.adapter = starAdapter
            sp.setSelection(0)
            sp.onItemSelectedListener = MySelectedListener()
        }

        private val starArray = arrayOf("水星", "金星", "地球", "火星", "木星", "土星")
        internal inner class MySelectedListener : OnItemSelectedListener {
            override fun onItemSelected(arg0: AdapterView<*>, arg1: View, arg2: Int,
```

```
arg3: Long) {
                toast("你选择的行星是${starArray[arg2]}")
            }

            override fun onNothingSelected(arg0: AdapterView<*>) {}
        }
```

瞧瞧，号称终结者的 Kotlin 也不过尔尔，整体代码量跟 Java 相比是半斤八两，丝毫不见往日的威风。由于这里的 Java 代码逻辑实在拐弯抹角，既有数组适配器又有选择监听器，因此 Kotlin 确实没有好办法。既然此路不通，那就试试别的办法，前面提到 Spinner 其实由两部分组成，一部分是直接显示在界面上的带箭头文本，另一部分是点击文本后弹出的选择对话框，所以能不能绕过 Spinner，干脆把下拉框分离成两个控件。倘若仅仅是一个带箭头的文本，毫无疑问使用文本视图 TextView 就可以了，箭头图标可以在布局文件中通过 drawableRight 属性来指定。

于是布局文件中的 Spinner 节点如下所示：

```
        <Spinner
            android:id="@+id/sp_dialog"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_toRightOf="@+id/tv_dialog"
            android:gravity="left|center"
            android:spinnerMode="dialog" />
```

表面上完全可以被下面这个 TextView 节点所取代：

```
        <TextView
            android:id="@+id/tv_spinner"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_toRightOf="@+id/tv_dialog"
            android:gravity="center"
            android:drawableRight="@drawable/arrow_down"
            android:textColor="@color/black"
            android:textSize="17sp" />
```

如果再来一个选择对话框，这样只要给该文本视图添加点击事件，点击 TextView 便弹出选择框，岂不是万事大吉？正巧 Anko 库已经提供了这股东风，它便是 selector，与 alert 一样来自于 Context 的扩展函数，调用格式形如“selector(对话框标题, 字符串队列) { i -> 第 i 项的选中处理代码 }”。那么将 selector 与前面的文本视图相结合，即可无缝实现原来的下拉框功能，具体的 Kotlin 代码如下所示：

```
        val satellites = listOf("水星", "金星", "地球", "火星", "木星", "土星")
        tv_spinner.text = satellites[0]
        tv_spinner.setOnClickListener {
            selector("请选择行星", satellites) { i ->
                tv_spinner.text = satellites[i]
```

```
            toast("你选择的行星是${tv_spinner.text}")
        }
    }
```

看看这几行代码，完全不见了数组适配器和选择监听器的踪影，故而代码量一下剧减到对应 Java 代码的三分之一。当然，为了正常地使用 selector 函数，不要忘了在代码文件头部加入下面一行导入语句：

```
import org.jetbrains.anko.selector
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
        compile "org.jetbrains.anko:anko-common:$anko_version"
```

虽然把布局文件里面的 Spinner 控件换成 TextView 控件，但是二者在功能上是没什么区别的，同样支持点击文本弹出选择框，也同样支持选中某项的回调。改造后下拉框的完整调用效果如图 7-1～图 7-3 所示，其中图 7-1 所示为界面上的下拉框初始文本，图 7-2 所示为点击文本后弹出的选择对话框，图 7-3 所示为选中某项后的下拉框界面。



图 7-1　下拉框的初始界面



图 7-2　点击下拉框弹出的选择对话框

如此方便易用的 selector，竟然撇开了数组适配器和选择监听器，那么它又是怎么实现的呢？认真阅读 Anko 库里面的 selector 源码，发现原来该函数利用了 AlertDialog 的 setItems 方法，通过 setItems 方法指定一串文本，并且定义了每项的点击事件，其运行结果竟然与 Spinner 的选择对话框殊途同归。



图 7-3　选中某项的下拉框

下面给出 AlertDialog 对应 selector 函数的 Java 实现代码，方便读者理解它的本质：

```
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle("请选择行星");
    builder.setItems(satellites, new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
```

```
                Toast.makeText(SpinnerDialogActivity.this, "你选择的行星是
"+starArray[arg2], Toast.LENGTH_LONG).show();
            }
        });
        builder.create().show();
```

7.1.2　列表视图 ListView

7.1.1 小节介绍了下拉框只有在弹出选择对话框时才会显示列表信息，一旦回到主界面，则仅仅展示选中的记录信息。相比之下，列表视图 ListView 允许将整个列表信息搬到主界面上，使得页面内容既规整又丰富，故而列表视图常用于展现新闻列表、商品列表、书籍列表等，方便用户逐行浏览与操作。

为实现各种排列组合类的视图（包括但不限于下拉框 Spinner、下拉列表 ListView、网格视图 GridView 等），Android 提供了五花八门的适配器用于组装某个规格的数据，常见的适配器有数组适配器 ArrayAdapter、简单适配器 SimpleAdapter、基本适配器 BaseAdapter、翻页适配器 PagerAdapter。适配器的种类虽多，却个个都不好用，以数组适配器为例，它与 Spinner 配合实现下拉框效果，其实现代码纷复繁杂，一直为人所诟病。故而在下拉框小节之中，干脆把 ArrayAdapter 连同 Spinner 一股脑都摒弃了，取而代之的是 Kotlin 扩展函数 selector。

到了列表视图 ListView 这里，与之搭档的一般是基本适配器 BaseAdapter，这个 BaseAdapter 更不简单，基于它的列表适配器得重写好几个方法，还有那个想让初学者撞墙的视图持有者 ViewHolder。总之，每当要实现类似新闻列表、商品列表之类的页面，一想到这个难缠的 BaseAdapter，心里便发怵。譬如图 7-4 所示的六大行星说明列表，左侧为图标，右侧为文字说明，其实是一个很普通的页面。

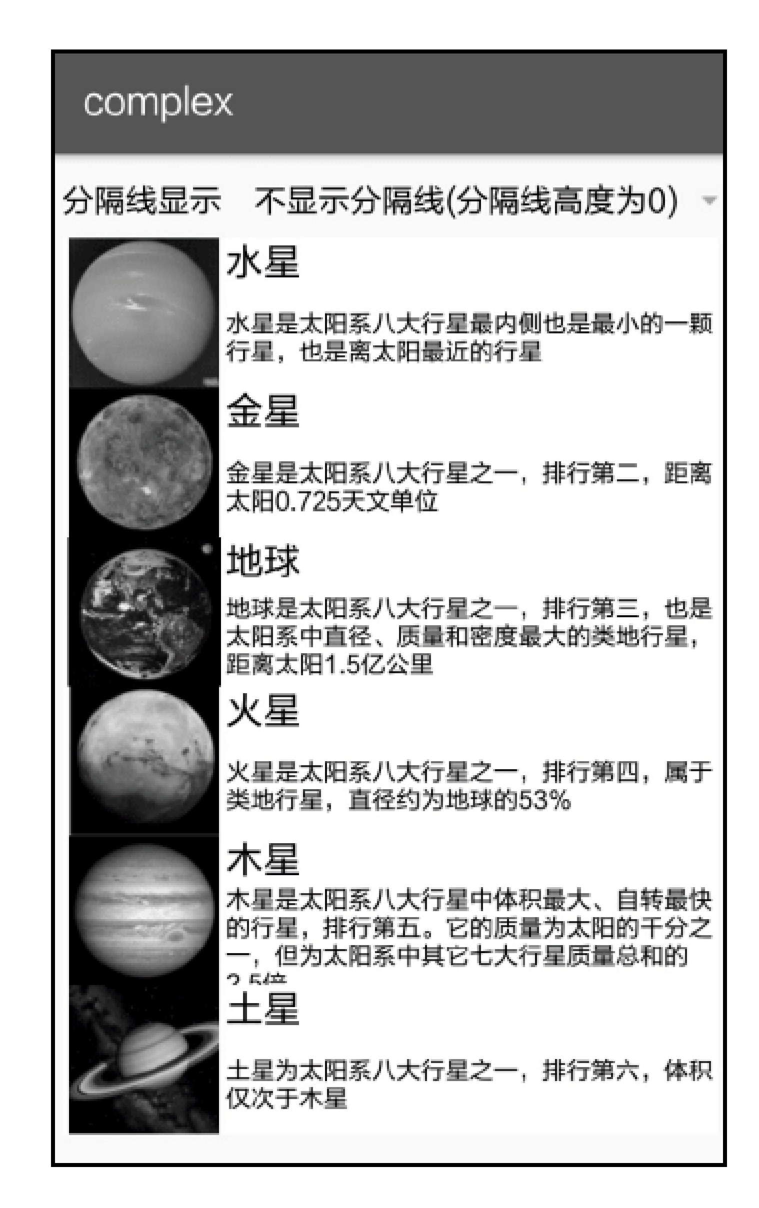


图 7-4　没有分隔线的行星列表

可是这个行星列表页面倘若使用 Java 编码，就得书写下面一大段适配器代码：

```
    public class PlanetJavaAdapter extends BaseAdapter  {
        private Context mContext;
        private ArrayList<Planet> mPlanetList;
        private int mBackground;

        public PlanetJavaAdapter(Context context, ArrayList<Planet> planet_list,
int background) {
            mContext = context;
            mPlanetList = planet_list;
            mBackground = background;
        }
```

```
        @Override
        public int getCount() {
            return mPlanetList.size();
        }

        @Override
        public Object getItem(int arg0) {
            return mPlanetList.get(arg0);
        }

        @Override
        public long getItemId(int arg0) {
            return arg0;
        }

        @Override
        public View getView(final int position, View convertView, ViewGroup parent) {
            ViewHolder holder = null;
            if (convertView == null) {
                holder = new ViewHolder();
                convertView =
LayoutInflater.from(mContext).inflate(R.layout.item_list_view, null);
                holder.ll_item = (LinearLayout)
convertView.findViewById(R.id.ll_item);
                holder.iv_icon = (ImageView)
convertView.findViewById(R.id.iv_icon);
                holder.tv_name = (TextView)
convertView.findViewById(R.id.tv_name);
                holder.tv_desc = (TextView)
convertView.findViewById(R.id.tv_desc);
                convertView.setTag(holder);
            } else {
                holder = (ViewHolder) convertView.getTag();
            }
            Planet planet = mPlanetList.get(position);
            holder.ll_item.setBackgroundColor(mBackground);
            holder.iv_icon.setImageResource(planet.image);
            holder.tv_name.setText(planet.name);
            holder.tv_desc.setText(planet.desc);
            return convertView;
        }

        public final class ViewHolder {
            public LinearLayout ll_item;
            public ImageView iv_icon;
            public TextView tv_name;
```

```
            public TextView tv_desc;
        }
    }
```

上面 Java 代码实现的适配器类 PlanetJavaAdapter 果真是既冗长又晦涩，然而这段代码模板基本上是列表视图的标配，只要用 Java 编码，就必须依样画瓢。如果用 Kotlin 实现这个适配器类会是怎样的呢？马上利用 Android Studio 把上述 Java 代码转换为 Kotlin 编码，转换后的 Kotlin 代码类似以下片段：

```
    class PlanetKotlinAdapter(private val mContext: Context, private val
mPlanetList: ArrayList<Planet>, private val mBackground: Int) : BaseAdapter() {

        override fun getCount(): Int {
            return mPlanetList.size
        }

        override fun getItem(arg0: Int): Any {
            return mPlanetList[arg0]
        }

        override fun getItemId(arg0: Int): Long {
            return arg0.toLong()
        }

        override fun getView(position: Int, convertView: View?, parent: ViewGroup):
View {
            var view = convertView
            var holder: ViewHolder?
            if (view == null) {
                holder = ViewHolder()
                view =
LayoutInflater.from(mContext).inflate(R.layout.item_list_view, null)
                holder.ll_item = view.findViewById<View>(R.id.ll_item) as
LinearLayout
                holder.iv_icon = view.findViewById<View>(R.id.iv_icon) as ImageView
                holder.tv_name = view.findViewById<View>(R.id.tv_name) as TextView
                holder.tv_desc = view.findViewById<View>(R.id.tv_desc) as TextView
                view.tag = holder
            } else {
                holder = view.tag as ViewHolder
            }
            val planet = mPlanetList[position]
            holder.ll_item!!.setBackgroundColor(mBackground)
            holder.iv_icon!!.setImageResource(planet.image)
            holder.tv_name!!.text = planet.name
            holder.tv_desc!!.text = planet.desc
```

```
            return view!!
        }

        inner class ViewHolder {
            var ll_item: LinearLayout? = null
            var iv_icon: ImageView? = null
            var tv_name: TextView? = null
            var tv_desc: TextView? = null
        }
    }
```

相比之下，直接转换得来的 Kotlin 代码最大的改进是把构造函数及初始化参数放到了第一行，其他地方未有明显优化。眼瞅着没多大改善，反而因为 Kotlin 的空安全机制平白无故多了好些问号和双感叹号，可谓得不偿失。问题出在 Kotlin 要求每个变量都要初始化上面，视图持有者 ViewHolder 作为一个内部类，目前虽然无法直接对控件对象赋值，但是从代码逻辑可以看出，适配器先从布局文件获取控件，然后才会调用各种设置方法。这意味着，上面的控件对象必定是先获得实例，在它们被使用的时候肯定是非空的，因此完全可以告诉编译器，这些控件对象一定会在使用前赋值，编译器您老就高抬贵手，睁一只眼闭一只眼放行好了。

毋庸置疑，该想法合情合理，Kotlin 正好提供了这种后门，它便是修饰符 lateinit。lateinit 的意思是延迟初始化，把它放在 var 或者 val 前面，表示被修饰的变量属于延迟初始化属性，即使没有初始化也仍然是非空的。如此一来，这些控件在声明时无须赋空值，在使用的时候也不必画蛇添足加上两个感叹号。根据新来的 lateinit 修改前面的行星列表适配器，改写后的 Kotlin 适配器代码如下所示：

```
    //适配器的属性定义及其初始化操作在主构造函数中自动完成
    class PlanetListAdapter(private val context: Context, private val planetList:
MutableList<Planet>, private val background: Int) : BaseAdapter() {

        //利用简化函数直接用等号连接函数体
        override fun getCount(): Int = planetList.size

        override fun getItem(position: Int): Any = planetList[position]

        override fun getItemId(position: Int): Long = position.toLong()

        override fun getView(position: Int, convertView: View?, parent: ViewGroup):
View {
            var view = convertView
            val holder: ViewHolder
            if (convertView == null) {
                view = LayoutInflater.from(context).inflate
(R.layout.item_list_view, null)
                holder = ViewHolder()
                //先声明视图持有者的实例，再依次获取内部的各个控件对象
                //findViewById 后面直接跟上"<视图类型>"，即可起到关键字 as 强制转换类型的功能
                holder.ll_item = view.findViewById<LinearLayout>(R.id.ll_item)
```

```
            holder.iv_icon = view.findViewById<ImageView>(R.id.iv_icon)
            holder.tv_name = view.findViewById<TextView>(R.id.tv_name)
            holder.tv_desc = view.findViewById<TextView>(R.id.tv_desc)
            view.tag = holder
        } else {
            holder = view.tag as ViewHolder
        }
        val planet = planetList[position]
        //因为 ll_item 被声明为延迟初始化属性，所以编译器认为它是个非空变量，就无须添加
        holder.ll_item.setBackgroundColor(background)
        holder.iv_icon.setImageResource(planet.image)
        holder.tv_name.text = planet.name
        holder.tv_desc.text = planet.desc
        return view!!
    }

    //ViewHolder 中的属性使用关键字 lateinit 延迟初始化
    inner class ViewHolder {
        lateinit var ll_item: LinearLayout
        lateinit var iv_icon: ImageView
        lateinit var tv_name: TextView
        lateinit var tv_desc: TextView
    }
}
```

以上的 Kotlin 代码总算有点模样了，虽然总体代码还不够精简，但是至少清晰明了，其中主要运用了 Kotlin 的以下三项技术：

（1）构造函数和初始化参数放在类定义的首行，无须单独构造，也无须手工初始化。

（2）像 getCount、getItem、getItemId 这三个函数，仅仅返回简单运算的数值，可以直接用等号取代大括号。

（3）对于视图持有者的内部控件，在变量名称前面添加修饰符 lateinit，表示该属性为延迟初始化属性。

外部给列表视图设置对应的适配器，直接给 adapter 属性赋值即可。下面是设置列表视图适配器的 Kotlin 代码例子：

```
        lv_planet.adapter = PlanetListAdapter(this, Planet.defaultList,
Color.WHITE)
```

上面的列表视图展示行星列表的效果如图 7-5 和图 7-6 所示，其中图 7-5 所示为只存在内部分割线时的行星列表界面，图 7-6 所示为存在包括头尾在内所有分隔线时的行星列表界面。

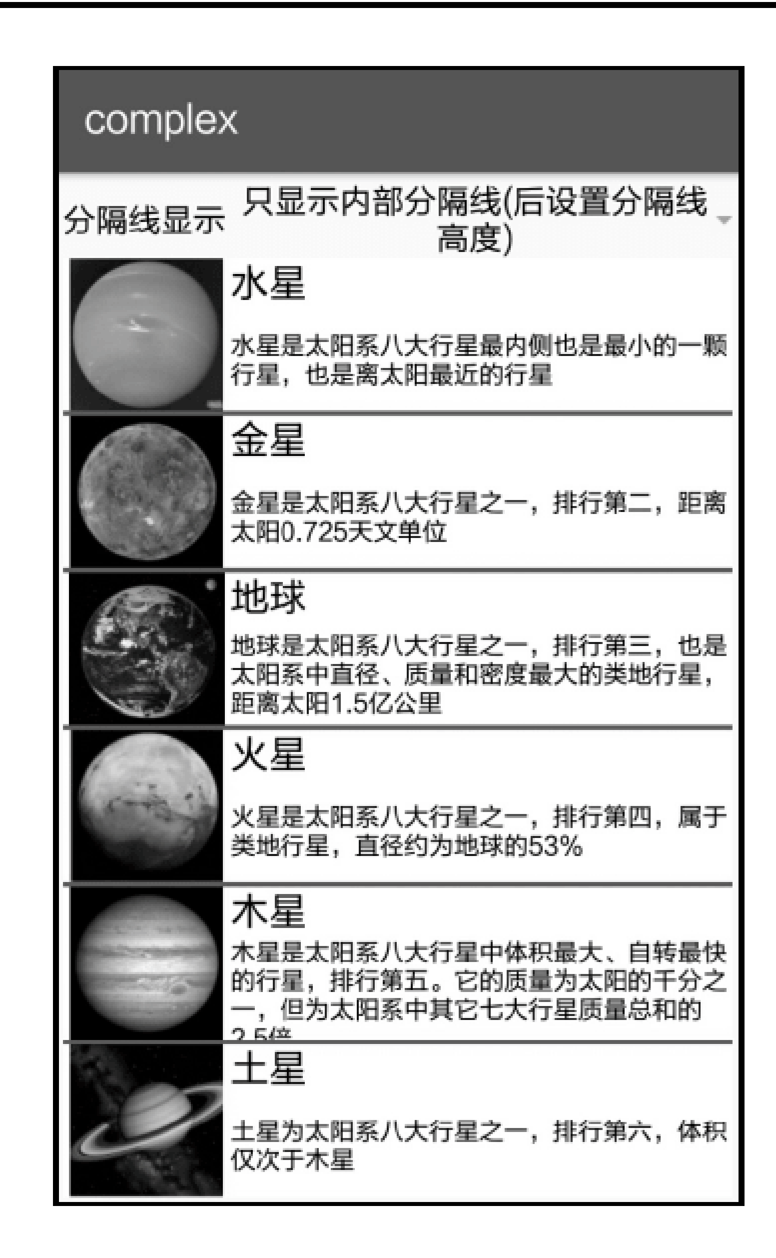


图 7-5　只有内部分隔线的行星列表

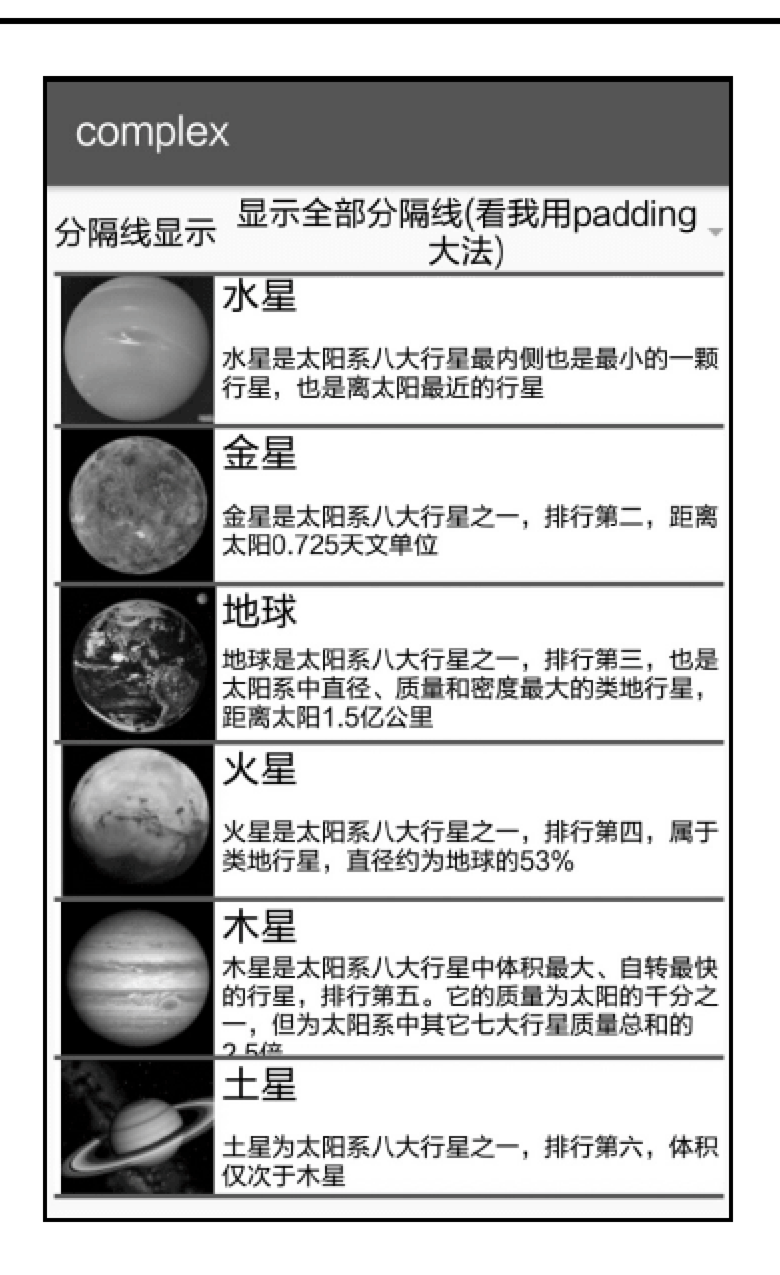


图 7-6　拥有头尾分隔线的行星列表

7.1.3　网格视图 GridView

除了列表视图外，网格视图 GridView 也是一类常见的适配器视图，它用于分行分列显示表格信息，可以达到更紧凑的界面效果，因而比 ListView 更适合展示商品清单。

网格视图同样需要通过适配器展示网格数据，7.1.2 小节 ListView 使用的基本适配器 BaseAdapter 也能用于 GridView。在前面的列表视图代码中，给出了 Kotlin 改写后的适配器类 PlanetListAdapter，其中通过修饰符 lateinit 固然避免了麻烦的空校验，可是控件对象迟早要初始化，晚赋值不如早赋值。翻到前面适配器 PlanetListAdapter 的实现代码，认真观察发现控件对象的获取其实依赖于布局文件的视图对象 view。既然如此，不妨把该视图对象作为 ViewHolder 的构造参数传过去，使得视图持有者在构造时便能一起初始化内部的控件对象。

据此，改写后的 Kotlin 适配器代码如下所示：

```
    //适配器的属性定义及其初始化操作在主构造函数中自动完成
    class PlanetGridAdapter(private val context: Context, private val planetList: MutableList<Planet>, private val background: Int) : BaseAdapter() {

        //利用简化函数直接用等号连接函数体
        override fun getCount(): Int = planetList.size

        override fun getItem(position: Int): Any = planetList[position]

        override fun getItemId(position: Int): Long = position.toLong()
```

```
        override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
            var view = convertView
            val holder: ViewHolder
            if (view == null) {
                view = LayoutInflater.from(context).inflate(R.layout.item_grid_view, null)
                holder = ViewHolder(view)
                //视图持有者的内部控件对象已经在构造时一并初始化了，故这里无须再做赋值
                view.tag = holder
            } else {
                holder = view.tag as ViewHolder
            }
            val planet = planetList[position]
            //ll_item 在构造时就被初始化，理所当然是个非空变量
            holder.ll_item.setBackgroundColor(background)
            holder.iv_icon.setImageResource(planet.image)
            holder.tv_name.text = planet.name
            holder.tv_desc.text = planet.desc
            return view!!
        }

        //ViewHolder 中的属性在构造时初始化
        inner class ViewHolder(val view: View) {
            //findViewById 后面直接跟上“<视图类型>”，即可起到关键字 as 强制转换类型的功能
            val ll_item: LinearLayout = view.findViewById<LinearLayout>(R.id.ll_item)
            val iv_icon: ImageView = view.findViewById<ImageView>(R.id.iv_icon)
            val tv_name: TextView = view.findViewById<TextView>(R.id.tv_name)
            val tv_desc: TextView = view.findViewById<TextView>(R.id.tv_desc)
        }
    }
```

外部若要调用改进后的网格适配器，直接给网格视图的 adapter 属性赋值即可。下面是设置网格视图适配器的 Kotlin 代码例子：

```
        gv_planet.adapter = PlanetGridAdapter(this, Planet.defaultList, Color.WHITE)
```

接着运行测试应用，得到的行星网格效果如图 7-7 和图 7-8 所示，其中图 7-7 所示为不存在分割线时的行星网格界面，图 7-8 所示为存在分隔线时的行星网格界面。

至此，基于 BaseAdapter 的 Kotlin 列表/网格适配器告一段落，上述的适配器代码模板同时适用于列表视图 ListView 与网格图 GridView。

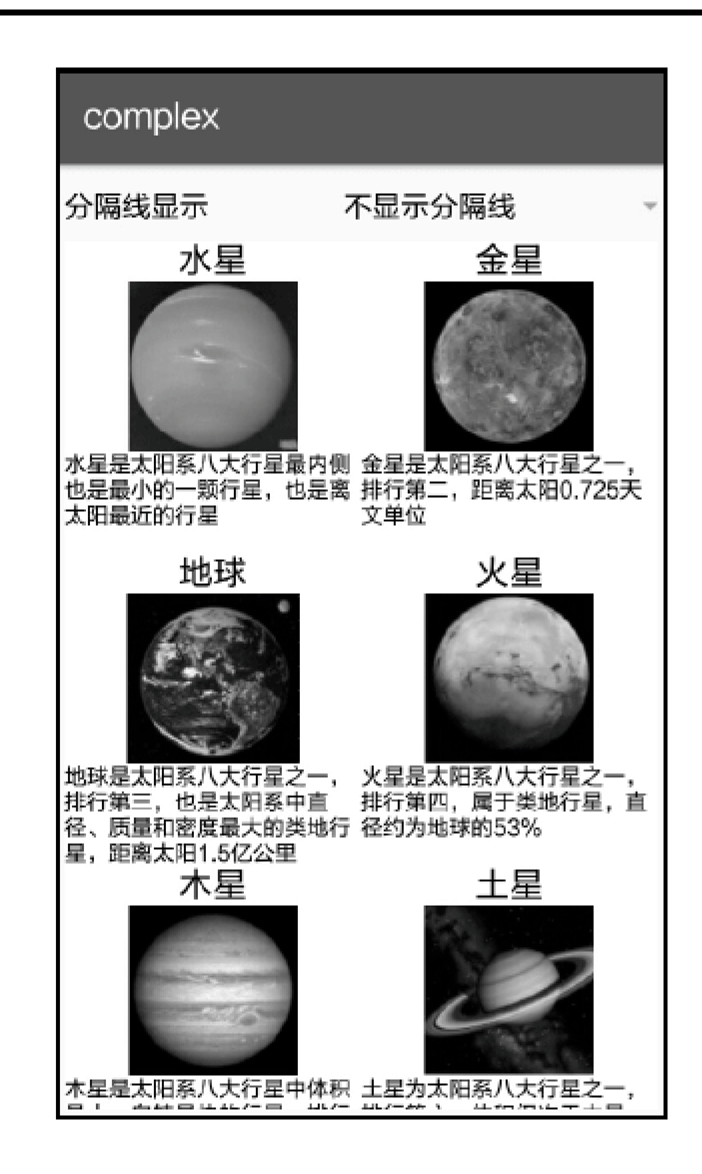


图 7-7　没有分隔线的行星网格

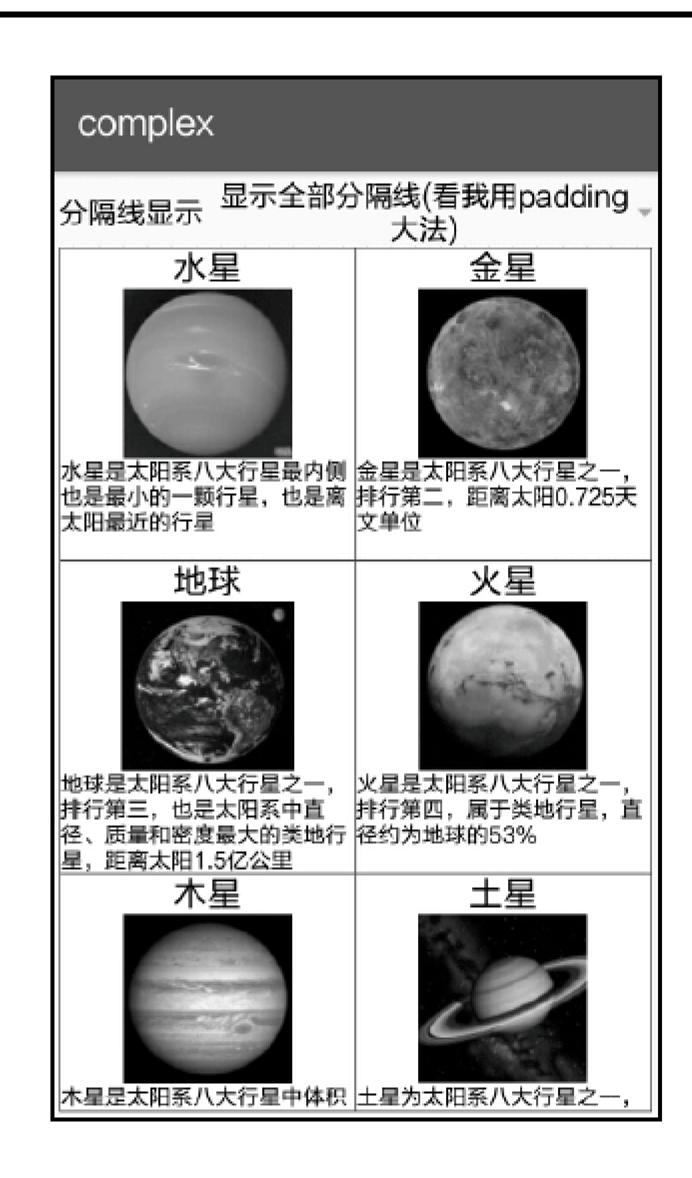


图 7-8　拥有分隔线的行星网格

7.1.4　循环视图 RecyclerView

循环视图是一种功能更加强大的列表类视图，它能够实现三种列表形式：线性列表、网格列表、瀑布流网格。由于 RecyclerView 是 Android 5.0 之后的新增控件，因此为了兼容以前的 Adnroid 版本，在使用该控件前要修改 build.gradle，在 dependencies 节点中加入下面一行表示导入 recyclerview 库：

```
//需要将“$supportVersion”替换为读者电脑上的 v7 库版本号
compile "com.android.support:recyclerview-v7:$supportVersion"
```

循环视图之所以功能强大，是因为它实现了一些特效处理，而不再仅限于简单的信息陈列。这些特效处理主要包括三点：更完备的方法调用、布局管理器以及循环适配器。

1. 更完备的方法调用

RecyclerView 提供了几个特效要求的方法，当然有部分方法在 Kotlin 中可转为属性操作，这些方法在 Kotlin 和 Java 之间的调用方式对比见表 7-1。

表 7-1　循环视图有关方法/属性的 Kotlin 和 Java 调用方式对比

方法/属性说明	Kotlin 的调用方式	Java 的调用方式
设置列表项的布局管理器	layoutManager	setLayoutManager
设置列表项的适配器	adapter	setAdapter
设置列表项的增删动画	itemAnimator	setItemAnimator
添加列表项的分隔线	addItemDecoration	addItemDecoration

2. 布局管理器

布局管理器 LayoutManager 是 RecyclerView 的精髓，也是 RecyclerView 之所以强悍的源泉。

它不但提供了三类布局管理，分别实现类似列表视图、网格视图、瀑布流网格的效果，而且可在代码中随时由循环视图对象通过 layoutManager 属性设置新的布局，一旦设置了 layoutManager 属性，界面就会根据新布局刷新列表项。这个特性尤其适合于手机在竖屏与横屏之间的显示切换（如竖屏时展示列表，横屏时展示网格），也适合在不同屏幕分辨率（如手机与平板）之间的显示切换（如在手机上展示列表，在平板上展示网格）。下面对这三类布局管理器分别进行概要介绍。

（1）线性布局管理器 LinearLayoutManager

LinearLayoutManager 类似于线性布局 LinearLayout，当它是垂直方向布局时，展示效果类似垂直的列表视图 ListView；当它是水平方向布局时，展示效果类似水平的列表视图。

（2）网格布局管理器 GridLayoutManager

GridLayoutManager 类似于网格布局 GridLayout（该控件是 Android 4.0 之后新加的控件），但从展示效果来看，GridLayoutManager 类似于网格视图 GridView。所以，开发者不用关心 GridLayout，就把 GridLayoutManager 当成 GridView 一样使用就好了。

（3）瀑布流网格布局管理器 StaggeredGridLayoutManager

电商 App 在展示众多商品信息时，往往使用高度灵活的高低格子来展示。因为不同商品的外观尺寸很不一样，比如冰箱是高高的在纵向上长，空调则是在横向上长，所以若用一样规格的网格来展示，必然有的商品图片被压缩得很小。这种情况下得根据不同的商品形状来展示不同高度的图片，这就是瀑布流网格的应用场合。StaggeredGridLayoutManager 让瀑布流效果的开发大大简化了，开发者只要在适配器中动态设置每个网格的高度，系统便会自动在界面上依次排列瀑布流网格。

3. 循环适配器

循环视图有专门的适配器类，即循环适配器 RecyclerView.Adapter。在设置 RecyclerView 的 adapter 属性之前，要先实现一个从 RecyclerView.Adapter 派生而来的数据适配器，用来定义列表项的布局与具体操作。总的来说，RecyclerView.Adapter 与前面遇到的 BaseAdapter 在处理流程上是基本一致的，当然它们之间也有不小的差异，下面是循环适配器和其他适配器的主要区别：

（1）自带视图持有者 ViewHolder 及其重用功能，无须开发者手工重用 ViewHolder。

（2）未自带列表项的点击和长按功能，需要开发者自己实现点击和长按事件的监听。

（3）增加区分不同列表项的视图类型，方便开发者根据类型加载不同的布局。

（4）可单独对个别项进行增删改操作，而无须刷新整个列表。

前面两小节在介绍列表视图和网格视图时，它们的适配器代码都存在视图持有者 ViewHolder，因为 Android 对列表类视图提供了回收机制，所以如果某些列表项在屏幕上看不到了，系统就会自动回收相应的视图对象。随着用户的下拉或者上拉手势，已经被回收的列表项会重新加载到界面上，倘若每次加载列表项都得从头创建视图对象，势必增加系统的资源开销。所以 ViewHolder 便应运而生，它在列表项首次初始化时，就将其视图对象保存起来，后面再次加载该视图时，即可直接从持有者处获得先前的视图对象，从而减少系统开销，提高系统的运行效率。

视图持有者的设计理念固然美好，却苦了 Android 开发者，因为每次由 BaseAdapter 派生新的适配器类，都必须手工处理视图持有者的相关逻辑，实在是个沉重的负担。鉴于此，循环视图适配器把视图持有者的重用逻辑剥离出来，由系统自行判断并处理持有者的重用操作。开发者编码继承

RecyclerView.Adapter 之后，只要完成业务上的代码逻辑即可，无须进行类似 BaseAdapter 视图持有者的手工重用。

现在由 Kotlin 实现循环视图适配器，综合前面两小节提到的优化技术，加上视图持有者的自动重用，适配器代码又得到了进一步的精简。由于循环视图适配器并不提供列表项的点击事件，因此开发者要自己编写包括点击、长按在内的事件处理代码。为方便理解循环适配器的 Kotlin 编码，下面以微信公众号消息列表为例给出对应的消息列表 Kotlin 代码：

```
    //ViewHolder 在构造时初始化布局中的控件对象
    class RecyclerLinearAdapter(private val context: Context, private val infos:
MutableList<RecyclerInfo>) : RecyclerView.Adapter<ViewHolder>(),
OnItemClickListener, OnItemLongClickListener {
        val inflater: LayoutInflater = LayoutInflater.from(context)

        //获得列表项的数目
        override fun getItemCount(): Int = infos.size

        //创建整个布局的视图持有者
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
            val view: View = inflater.inflate(R.layout.item_recycler_linear, parent,
false)
            return ItemHolder(view)
        }

        //绑定每项的视图持有者
        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val vh: ItemHolder = holder as ItemHolder
            vh.iv_pic.setImageResource(infos[position].pic_id)
            vh.tv_title.text = infos[position].title
            vh.tv_desc.text = infos[position].desc
            // 列表项的点击事件需要自己实现
            vh.ll_item.setOnClickListener { v ->
                itemClickListener?.onItemClick(v, position)
            }
            vh.ll_item.setOnLongClickListener { v ->
                itemLongClickListener?.onItemLongClick(v, position)
                true
            }
        }

        //ItemHolder 中的属性在构造时初始化
        inner class ItemHolder(view: View) : RecyclerView.ViewHolder(view) {
            var ll_item = view.findViewById<LinearLayout>(R.id.ll_item)
            var iv_pic = view.findViewById<ImageView>(R.id.iv_pic)
            var tv_title = view.findViewById<TextView>(R.id.tv_title)
```

```
            var tv_desc = view.findViewById<TextView>(R.id.tv_desc)
        }

        //自定义点击事件监听器
        private var itemClickListener: OnItemClickListener? = null
        fun setOnItemClickListener(listener: OnItemClickListener) {
            this.itemClickListener = listener
        }

        //自定义长按事件监听器
        private var itemLongClickListener: OnItemLongClickListener? = null
        fun setOnItemLongClickListener(listener: OnItemLongClickListener) {
            this.itemLongClickListener = listener
        }

        override fun onItemClick(view: View, position: Int) {
            val desc = "您点击了第${position+1}项，标题是${infos[position].title}"
            context.toast(desc)
        }

        override fun onItemLongClick(view: View, position: Int) {
            val desc = "您长按了第${position+1}项，标题是${infos[position].title}"
            context.toast(desc)
        }
    }
```

接下来，外部就可以对循环视图对象运用 Kotlin 版本的循环适配器了，具体的 Kotlin 调用代码如下所示：

```
            rv_linear.layoutManager = LinearLayoutManager(this)
            val adapter = RecyclerLinearAdapter(this, RecyclerInfo.defaultList)
            adapter.setOnItemClickListener(adapter)
            adapter.setOnItemLongClickListener(adapter)
            rv_linear.adapter = adapter
            rv_linear.itemAnimator = DefaultItemAnimator()
            rv_linear.addItemDecoration(SpacesItemDecoration(1))
```

以上的循环适配器代码初步实现了公众号消息列表的展示页面，具体的列表效果如图 7-9 所示。



图 7-9　仿微信公众号的消息列表

可是这个循环适配器 RecyclerLinearAdapter 仍然体量庞大，细细观察发现其实它有着数个与具体业务无关的属性与方法，譬如上下文对象 context、布局载入对象 inflater、点击监听器 itemClickListener、长按监听器 itemLongClickListener 等，故而完全可以把这些通用部分提取到一个基类，然后具体业务再从该基类派生出特定的业务适配器类。根据这种设计思路，提取出了循环视图基础适配器，它的 Kotlin 代码如下所示：

```
    //循环视图基础适配器
    abstract class RecyclerBaseAdapter<VH : RecyclerView.ViewHolder>(val context:
Context) : RecyclerView.Adapter<RecyclerView.ViewHolder>(), OnItemClickListener,
OnItemLongClickListener {
        val inflater: LayoutInflater = LayoutInflater.from(context)

        //获得列表项的个数，需要子类重写
        override abstract fun getItemCount(): Int

        //根据布局文件创建视图持有者，需要子类重写
        override abstract fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder

        //绑定视图持有者中的各个控件对象，需要子类重写
        override abstract fun onBindViewHolder(holder: RecyclerView.ViewHolder,
position: Int)

        override fun getItemViewType(position: Int): Int = 0

        override fun getItemId(position: Int): Long = position.toLong()

        var itemClickListener: OnItemClickListener? = null
        fun setOnItemClickListener(listener: OnItemClickListener) {
            this.itemClickListener = listener
        }

        var itemLongClickListener: OnItemLongClickListener? = null
        fun setOnItemLongClickListener(listener: OnItemLongClickListener) {
            this.itemLongClickListener = listener
        }

        override fun onItemClick(view: View, position: Int) {}

        override fun onItemLongClick(view: View, position: Int) {}
    }
```

一旦有了这个基础适配器，实际业务的适配器即可由此派生而来，真正需要开发者编写的代码一下精简了不少。下面便是一个循环视图的网格适配器，它实现了类似淘宝主页的网格频道栏目，具体的 Kotlin 代码如下所示：

```
    //把公共属性和公共方法剥离到基类 RecyclerBaseAdapter
    //此处仅需实现 getItemCount、onCreateViewHolder、onBindViewHolder 三个方法，以及视
图持有者的类定义
    class RecyclerGridAdapter(context: Context, private val infos:
MutableList<RecyclerInfo>) :
RecyclerBaseAdapter<RecyclerView.ViewHolder>(context) {

        override fun getItemCount(): Int = infos.size
```

```
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder {
            val view: View = inflater.inflate(R.layout.item_recycler_grid, parent,
false)
            return ItemHolder(view)
        }

        override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position:
Int) {
            val vh = holder as ItemHolder
            vh.iv_pic.setImageResource(infos[position].pic_id)
            vh.tv_title.text = infos[position].title
        }

        inner class ItemHolder(view: View) : RecyclerView.ViewHolder(view) {
            var ll_item = view.findViewById<LinearLayout>(R.id.ll_item)
            var iv_pic = view.findViewById<ImageView>(R.id.iv_pic)
            var tv_title = view.findViewById<TextView>(R.id.tv_title)
        }
    }
```

下面依旧由外部通过循环视图对象调用改进后的网格适配器，对应的 Kotlin 调用代码示例如下：

```
        rv_grid.layoutManager = GridLayoutManager(this, 5)
        val adapter = RecyclerGridAdapter(this, RecyclerInfo.defaultGrid)
        adapter.setOnItemClickListener(adapter)
        adapter.setOnItemLongClickListener(adapter)
        rv_grid.adapter = adapter
        rv_grid.itemAnimator = DefaultItemAnimator()
        rv_grid.addItemDecoration(SpacesItemDecoration(1))
```

改进后的循环网格适配器运行之后的界面效果如图 7-10 所示，无缝实现了原来需要数十行 Java 代码才能实现的功能。

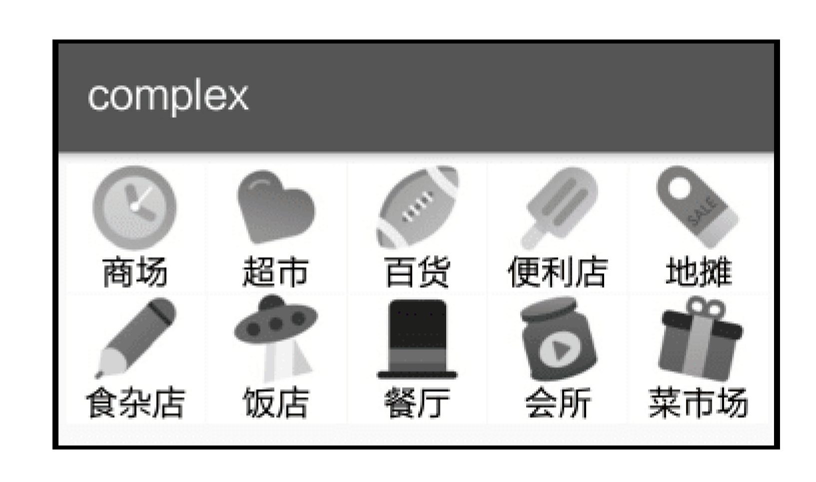


图 7-10　仿淘宝频道的分类网格

然而基类手段不过是雕虫小技，Java 也照样能够运用，所以这根本不入 Kotlin 的法眼，要想超越 Java，还得拥有独门秘籍才行。注意，适配器代码仍然通过 findViewById 方法获得控件对象，可是号称在 Anko 库的支持之下，Kotlin 早就无须该方法即可直接访问控件对象，为什么这里依旧靠老牛拉破车呢？其中的缘由是 Anko 库仅仅实现了 Activity 活动页面的控件自动获取，并未实现适配器内部的自动获取。当然，Kotlin 早就料到了这一手，为此专门提供了一个插件 LayoutContainer，只要开发者让自定义的 ViewHolder 类实现该接口，即可在视图持有者内部自动获取并直接使用控件对象。这下无论是在 Activity 代码还是在适配器代码中，均可将控件名称拿来直接调用。

这么神奇的魔法，快来看看优化后的 Kotlin 适配器代码是如何书写的：

```
    //利用 Kotlin 的插件 LayoutContainer 在适配器中直接使用控件对象，而无须对其进行显式声明
    class RecyclerStaggeredAdapter(context: Context, private val infos:
MutableList<RecyclerInfo>) : RecyclerBaseAdapter<RecyclerView.ViewHolder>
(context) {

        override fun getItemCount(): Int = infos.size

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder {
            val view: View = inflater.inflate(R.layout.item_recycler_staggered,
parent, false)
            return ItemHolder(view)
        }

        override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position:
Int) {
            (holder as ItemHolder).bind(infos[position])
        }

        //注意这里要去掉 inner，否则运行报错“java.lang.NoSuchMethodError: No virtual
method _$_findCachedViewById”
        class ItemHolder(override val containerView: View?) :
RecyclerView.ViewHolder(containerView), LayoutContainer {
            fun bind(item: RecyclerInfo) {
                //因为运用了插件 LayoutContainer，所以这里可以直接使用控件对象
                iv_pic.setImageResource(item.pic_id)
                tv_title.text = item.title
            }
        }
    }
```

当然，为了能够正常使用该功能，需要在适配器代码头部加上以下两行代码，其中第一行代码表示引用 Kotlin 的扩展插件 LayoutContainer，第二行代码表示导入指定布局文件里面所有控件对象：

```
    import kotlinx.android.extensions.LayoutContainer
    import kotlinx.android.synthetic.main.item_recycler_staggered.*
```

另外，因为 LayoutContainer 是 Kotlin 针对性提供给 Android 的扩展插件，所以需要修改模块的 build.gradle，在文件末尾添加下面几行配置，表示允许引用安卓插件库：

```
    //LayoutContainer 需要设置 experimental = true
    androidExtensions {
        experimental = true
    }
```

即使修改后的瀑布流适配器代码用到了新插件，外部仍旧同原来一样给循环视图设置适配器，以下的 Kotlin 调用代码并无任何变化：

```
            rv_staggered.layoutManager = StaggeredGridLayoutManager(3,
LinearLayout.VERTICAL)
            //第一种方式：使用采取 LayoutContainer 的插件适配器
            val adapter = RecyclerStaggeredAdapter(this, RecyclerInfo.defaultStag)
            rv_staggered.adapter = adapter
            rv_staggered.itemAnimator = DefaultItemAnimator()
            rv_staggered.addItemDecoration(SpacesItemDecoration(3))
```

上面采用了新的适配器插件，似乎已经大功告成，可是依然要书写单独的适配器代码，仔细研究发现这个 RecyclerStaggeredAdapter 还有三个要素是随着具体业务而变化的，参见如下说明：

（1）列表项的布局文件资源编码，如 R.layout.item_recycler_staggered。

（2）列表项信息的数据结构名称，如 RecyclerInfo。

（3）对各种控件对象的设置操作，如 ItemHolder 类的 bind 方法。

除了以上三个要素外，适配器 RecyclerStaggeredAdapter 内部的其余代码都是允许复用的，因此，接下来的工作就是想办法把这三个要素抽象为公共类的某种变量。对于第一个布局编码，可以考虑将其作为一个整型的输入参数；对于第二个数据结构，可以考虑定义一个模板类，在外部调用时再指定具体的数据类；对于第三个 bind 方法，若是 Java 编码早已束手无策，现用 Kotlin 编码正好将该方法作为一个函数参数传入。

依照上述三个要素的三种处理对策，进而提炼出来了循环适配器的通用工具类 RecyclerCommonAdapter，详细的 Kotlin 通用适配器定义代码示例如下：

```
    //循环视图通用适配器
    //将具体业务中会变化的三类要素抽取出来，作为外部传进来的变量。这三类要素包括：
    //布局文件对应的资源编号、列表项的数据结构、各个控件对象的初始化操作
    class RecyclerCommonAdapter<T>(context: Context, private val layoutId: Int,
private val items: List<T>, val init: (View, T) -> Unit):
RecyclerBaseAdapter<RecyclerCommonAdapter.ItemHolder<T>>(context) {

        override fun getItemCount(): Int = items.size

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder {
            val view: View = inflater.inflate(layoutId, parent, false)
            return ItemHolder<T>(view, init)
        }

        override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position:
Int) {
            val vh: ItemHolder<T> = holder as ItemHolder<T>
            vh.bind(items.get(position))
        }

        //注意 init 是一个函数形式的输入参数
        class ItemHolder<in T>(val view: View, val init: (View, T) -> Unit) :
RecyclerView.ViewHolder(view) {
```

```
            fun bind(item: T) {
                init(view, item)
            }
        }
    }
```

有了这个通用适配器，外部声明循环适配器只需像函数调用那样传入这三种变量就好了，具体的 Kotlin 调用代码如下所示：

```
        //第二种方式：使用把三类可变要素抽象出来的通用适配器
        val adapter = RecyclerCommonAdapter(this,
R.layout.item_recycler_staggered, RecyclerInfo.defaultStag,
                {view, item ->
                    val iv_pic = view.findViewById<ImageView>(R.id.iv_pic)
                    val tv_title = view.findViewById<TextView>(R.id.tv_title)
                    iv_pic.setImageResource(item.pic_id)
                    tv_title.text = item.title
                })
        rv_staggered.adapter = adapter
```

瞧瞧，最终出炉的瀑布流适配器仅有不到 10 行的代码，其中的关键技术——函数参数真是不鸣则已、一鸣惊人。至此，本节的适配器实现过程终于落下帷幕，一路上可谓是过五关斩六将，硬生生把数十行的 Java 代码压缩到不足 10 行的 Kotlin 代码，经过不断迭代优化方取得如此彪炳战绩。尤其是最后的两种实现方式，分别运用了 Kotlin 的多项综合技术，才能集 Kotlin 精妙语法之大成。这两种实现方式的瀑布流效果是一样的，具体演示用的服装列表界面如图 7-11 和图 7-12 所示，其中图 7-11 所示为服装列表的初始界面，图 7-12 所示为上拉列表之后的服装界面。



图 7-11　服装频道的瀑布流列表初始界面



图 7-12　瀑布流列表上拉之后的服装界面

7.2　使用材质设计 MaterialDesign

MaterialDesign 库是 Android 在界面设计方面做出重大提升的增强库，该库提供了协调布局 CoordinatorLayout、应用栏布局 AppBarLayout、可折叠工具栏布局 CollapsingToolbarLayout 等新颖控件，这几个新加的布局控件结合工具栏 Toolbar 能够实现导航栏动态伸缩的效果。本节就对头部导航栏动态特效的实现过程进行逐步地说明，最后总结起来叙述仿支付宝首页的头部伸缩动画效果。

7.2.1　协调布局 CoordinatorLayout

Android 自 5.0 之后对 UI 做了较大的提升，一个重大的改进是推出了 MaterialDesign 库，而该库的基础即为协调布局 CoordinatorLayout，几乎所有的 design 控件都依赖于该布局。所谓协调布局，指的是内部控件互相之间存在着动作关联，比如在 A 视图的位置发生变化时，B 视图的位置也按照某种规则来变化，仿佛弹钢琴有了协奏曲一般。

使用协调布局 CoordinatorLayout 时，要注意以下两点：

（1）需要给模块导入 design 库，即修改 build.gradle，在 dependencies 节点中加入下面一行表示导入 design 库：

```
//需要将“$supportVersion”替换为读者电脑上的 design 库版本号
compile 'com.android.support:design:$supportVersion'
```

（2）根布局采用 android.support.design.widget.CoordinatorLayout，且该节点要添加命名空间声明 xmlns:app="http://schemas.android.com/apk/res-auto"。使用协调布局的具体 XML 文件示例如下：

```
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/cl_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <!-- 此处省略内部的视图节点 -->
</android.support.design.widget.CoordinatorLayout>
```

协调布局 CoordinatorLayout 继承自 ViewGroup，它的实现效果类似于相对布局 RelativeLayout，若要指定子视图在整个页面中的位置，则有以下几个办法：

（1）使用 layout_gravity 属性，指定子视图在 CoordinatorLayout 内部的对齐方式。

（2）使用 app:layout_anchor 和 app:layout_anchorGravity 属性，指定子视图相对于其他子视图的位置。其中，app:layout_anchor 表示当前以哪个视图作为参照物，app:layout_anchorGravity 表示本视图相对于参照物的对齐方式。

（3）使用 app:layout_behavior 属性，指定子视图相对于其他视图的行为，当对方的位置发生变化时，本视图的位置也要随之变化。

接下来，为了说明协调布局的“协调”含义，先来看一个具体的例子，这个例子用到了悬浮按钮 FloatingActionButton。悬浮按钮是 design 库提供的一个特效按钮，它继承自图像按钮 ImageButton，除了图像按钮的所有功能之外，还提供了以下的额外功能：

（1）悬浮按钮会悬浮在其他视图之上，即使布局文件中别的视图在它后面，悬浮按钮也仍然显示在最前面。

（2）在隐藏和显示悬浮按钮时会播放切换动画，其中隐藏按钮操作调用了 hide 方法，显示按钮操作调用了 show 方法。

（3）悬浮按钮默认会随着便签条 Snackbar 的出现或消失而动态调整位置。

下面是演示协调布局中悬浮按钮 FloatingActionButton 与便签条 Snackbar 联动的布局文件例子：

```
    <android.support.design.widget.CoordinatorLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:id="@+id/cl_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <LinearLayout
            android:id="@+id/ll_main"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >

            <Button
                android:id="@+id/btn_snackbar"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center"
                android:layout_marginTop="30dp"
                android:text="显示简单提示条"
                android:textColor="@color/black"
                android:textSize="17sp" />

            <Button
                android:id="@+id/btn_floating"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center"
                android:layout_marginTop="30dp"
                android:text="隐藏悬浮按钮"
```

```
            android:textColor="@color/black"
            android:textSize="17sp" />
    </LinearLayout>

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab_btn"
        android:layout_width="80dp"
        android:layout_height="80dp"
        android:layout_margin="20dp"
        app:layout_anchor="@id/ll_main"
        app:layout_anchorGravity="bottom|right"
        android:background="@drawable/float_btn" />
</android.support.design.widget.CoordinatorLayout>
```

与上述布局对应的 Kotlin 演示代码很简单，仅仅在点击按钮时弹出便签条，调用代码如下所示：

```
        btn_snackbar.setOnClickListener {
            Snackbar.make(cl_main, "这是个提示条", Snackbar.LENGTH_LONG).show()
        }
```

由于便签条在屏幕底部弹出之后，短暂停留几秒便收缩消失，如此一进一出之间，即可观察悬浮按钮与便签条的协调联动。具体的悬浮按钮位置变化效果如图 7-13～图 7-15 所示，其中图 7-13 展示便签条弹出之前的界面，此时悬浮按钮位于屏幕右下方；图 7-14 展示便签条弹出之后的界面，此时悬浮按钮随着便签条一齐向上抬升一段距离；图 7-15 展示便签条回缩之后的界面，此时悬浮按钮跟着下移，并恢复到原来的屏幕位置。



图 7-13　便签条未弹出时的界面



图 7-14　便签条弹出之后的界面



图 7-15　便签条回缩之后的界面

7.2.2　工具栏 Toolbar

主流 App 除了底部有一排标签栏之外，通常顶部还有一排导航栏，在 Android 5.0 之前，这个顶部导航栏是以 ActionBar 控件的形式出现，但 ActionBar 存在着不灵活、难以扩展等毛病，所以 Android 5.0 之后推出了 Toolbar 工具栏控件，意在取代 ActionBar。

不过为了兼容之前的版本，原有的 ActionBar 控件仍然保留，可是 Toolbar 与 ActionBar 都占着顶部导航栏的位置，所以要想引入 Toolbar 就得先关闭 ActionBar。具体的替换操作步骤如下：

步骤 01 在 styles.xml 中定义一个不包含 ActionBar 的风格样式，代码例子如下所示：

```
        <style name="AppCompatTheme" parent=
"Theme.AppCompat.Light.NoActionBar" />
```

步骤 02 修改 AndroidManifest.xml，把 activity 节点的 android:theme 属性值改为第一步定义的风格，如 android:theme="@style/AppCompatTheme"。

步骤 03 把页面布局文件的根节点改为 LinearLayout，且为 vertical 垂直方向；然后增加一个 Toolbar 节点，因为 Toolbar 本质是一个 ViewGroup，所以也可以在它下面添加别的控件。下面是一个 Toolbar 节点的布局例子片段：

```
<android.support.v7.widget.Toolbar
    android:id="@+id/tl_head"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

步骤 04 Activity 代码需要继承 AppCompatActivity，其实在 Android Studio 中新建模块已经是默认继承 AppCompatActivity 了。

步骤 05 最后在 onCreate 函数中获取布局文件中的 Toolbar 对象，并调用 setSupportActionBar 方法设置当前的 Toolbar 对象。

工具栏 Toolbar 之所以比 ActionBar 灵活，除了允许开发者自行添加下级控件之外，还有一个原因是它提供了多个方法和属性来指定自带控件的风格。下面是使用 Toolbar 设置控件风格的 Kotlin 代码片段：

```
class ToolbarActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_toolbar)
        //设置工具栏的主标题文本内容
        tl_head.title = "这是工具栏的主标题"
        //设置工具栏的主标题文本颜色
        tl_head.setTitleTextColor(Color.RED)
        //设置工具栏左边的 Logo 图标
        tl_head.setLogo(R.drawable.ic_launcher)
        //设置工具栏的副标题文本内容
        tl_head.subtitle = "这是副标题"
        //设置工具栏的副标题文本颜色
        tl_head.setSubtitleTextColor(Color.YELLOW)
        //设置工具栏的背景
        tl_head.setBackgroundResource(R.color.blue_light)
        //使用 Toolbar 替换系统自带的 ActionBar
        setSupportActionBar(tl_head)
        //工具栏最左侧的导航图标，通常用作返回按钮
        tl_head.setNavigationIcon(R.drawable.ic_back)
        //最左侧导航图标的点击事件，即返回上一个页面
        //该方法必须放到 setSupportActionBar 之后，不然不起作用
        tl_head.setNavigationOnClickListener { finish() }
    }
}
```

具体的工具栏演示效果如图 7-16 所示，该工具栏的界面元素包括导航图标、工具栏图标、标题、副标题等。

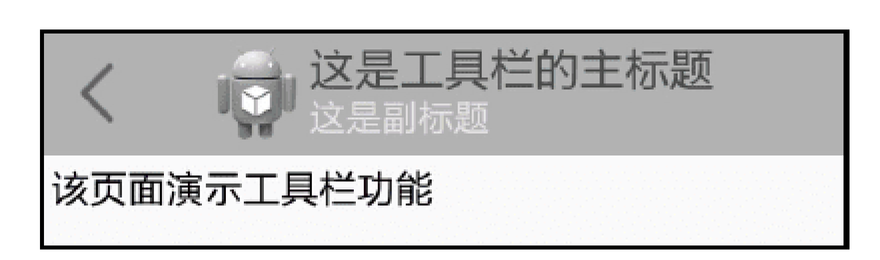


图 7-16　工具栏的演示界面

7.2.3　应用栏布局 AppBarLayout

前面提到 Android 推出工具栏 Toolbar 用来替代 ActionBar，使得导航栏的灵活性和易用性大大增强。可是仅仅使用 Toolbar，还是有些呆板，比如说 Toolbar 固定占据着页面顶端，既不能跟着页面主体移上去，也不会跟着页面主体拉下来。为了让 App 页面更加生动活泼，势必要求 Toolbar 在某些特定的场景上移或者下拉，如此才能满足酷炫的页面特效需求。为此，Android 5.0 推出了 MaterialDesign 库，通过该库中的协调布局和本小节要介绍的应用栏布局 AppBarLayout，将这两种布局结合起来对 Toolbar 加以包装，从而实现顶部导航栏的动态变化效果。

应用栏布局 AppBarLayout 其实继承自线性布局 LinearLayout，所以它具备 LinearLayout 的所有属性与方法，除此之外，应用栏布局的额外功能主要有以下几点：

（1）支持响应页面主体的滑动行为，即在页面主体进行上移或者下拉时，AppBarLayout 能够捕捉到页面主体的滚动操作。

（2）捕捉到滚动操作之后，还要通知头部控件（通常是 Toolbar），告诉头部控件你要怎么滚，是爱怎么滚就怎么滚，还是满大街滚。

顶部导航栏的动态滚动效果具体到实现上，则要在 App 工程中做如下修改：

（1）在 build.gradle 中添加几个库的编译支持，包括 appcompat-v7 库（Toolbar 需要）、design 库（AppBarLayout 需要）、recyclerview 库（主页面的 RecyclerView 需要）。

（2）布局文件的根布局采用 CoordinatorLayout，因为 design 库的动态效果都依赖于该控件，并且该节点要添加命名空间声明 xmlns:app="http://schemas.android.com/apk/res-auto"。

（3）使用 AppBarLayout 节点包裹 Toobar 节点，也就是将 Toobar 节点作为 AppBarLayout 节点的下级节点。

（4）给 Toobar 节点添加滚动属性 app:layout_scrollFlags="scroll|enterAlways"，指定工具栏的滚动行为标志。

（5）演示界面的页面主体使用 RecyclerView 控件，并给该控件节点添加行为属性，即 app:layout_behavior="@string/appbar_scrolling_view_behavior"，表示通知 AppBarLayout 捕捉 RecyclerView 的滚动操作。

下面是 AppBarLayout 结合 RecyclerView 的布局文件例子：

```
    <android.support.design.widget.CoordinatorLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
        android:id="@+id/cl_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <android.support.design.widget.AppBarLayout
            android:id="@+id/abl_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

            <android.support.v7.widget.Toolbar
                android:id="@+id/tl_title"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                android:background="@color/blue_light"
                app:layout_scrollFlags="scroll|enterAlways" />
        </android.support.design.widget.AppBarLayout>

        <android.support.v7.widget.RecyclerView
            android:id="@+id/rv_main"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    </android.support.design.widget.CoordinatorLayout>
```

与上述布局文件对应的 Kotlin 页面代码如下所示：

```
    class AppbarRecyclerActivity : AppCompatActivity() {
        private val yearArray = arrayOf("鼠年", "牛年", "虎年", "兔年", "龙年", "蛇
年", "马年", "羊年", "猴年", "鸡年", "狗年", "猪年")

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_appbar_recycler)
            setSupportActionBar(tl_title)
            rv_main.layoutManager = LinearLayoutManager(this)
            rv_main.adapter = RecyclerCollapseAdapter(this, yearArray)
        }
    }
```

应用栏布局配合循环视图的演示效果如图 7-17～图 7-19 所示，其中图 7-17 展示打开演示页的初始界面，此时工具栏位于页面顶部；图 7-18 展示上拉一小段时的界面，此时工具栏随着向上滚动一段；图 7-19 展示上拉一大段时的界面，此时工具栏滚动到屏幕之外，完全看不见了。



图 7-17　应用栏搭配循环视图的初始演示界面

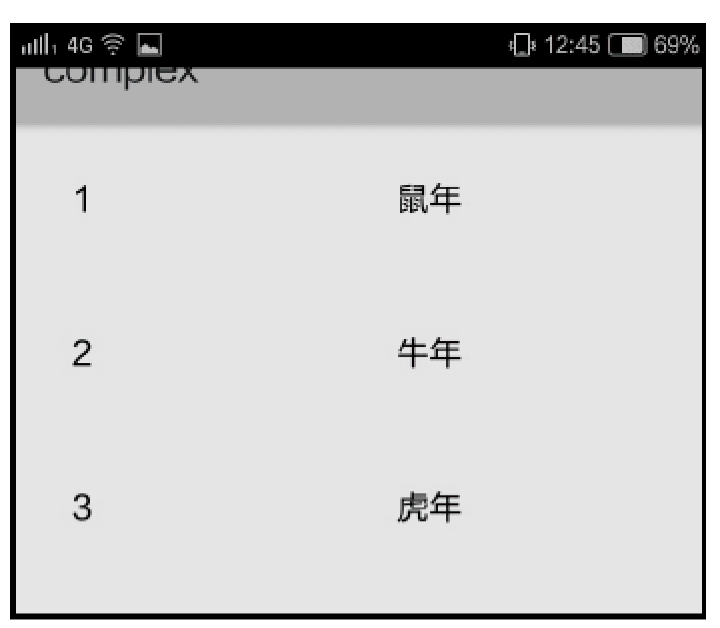


图 7-18　应用栏上拉一小段时的循环视图界面



图 7-19　应用栏上拉一大段时的循环视图界面

虽然通过 AppBarLayout 能够实现 Toolbar 的滚动效果，但并非所有可滚动的控件都会触发 Toolbar 滚动，事实上只有 Android 5.0 之后新增的少数滚动控件才具备该特技。RecyclerView 是身怀的绝技之一，它可用来替代列表视图 ListView 和网格视图 GridView；而替代滚动视图 ScrollView 的另有其人，它便是嵌套滚动视图 NestedScrollView，在 Android 5.0 之后的 v4 库中提供。

NestedScrollView 继承自框架布局 FrameLayout，其用法与 ScrollView 相似，例如都必须且只能带一个直接子视图，都允许内部视图上下滚动等。NestedScrollView 多出来的功能则是跟 AppBarLayout 配合使用，借由触发 Toolbar 的滚动行为，可把它当作兼容 Android 5.0 新特性的增强版 ScrollView。

下面是 AppBarLayout 结合 NestedScrollView 的布局文件例子：

```
    <android.support.design.widget.CoordinatorLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:id="@+id/cl_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <android.support.design.widget.AppBarLayout
            android:id="@+id/abl_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

            <android.support.v7.widget.Toolbar
                android:id="@+id/tl_title"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_scrollFlags="scroll|enterAlways"
                android:background="@color/blue_light" />
        </android.support.design.widget.AppBarLayout>

        <android.support.v4.widget.NestedScrollView
            android:id="@+id/nsv_main"
            android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                android:layout_width="match_parent"
                android:layout_height="100dp"
                android:background="#ffaaaa"
                android:gravity="center"
                android:text="hello"
                android:textColor="#000000"
                android:textSize="17sp" />

            <TextView
                android:layout_width="match_parent"
                android:layout_height="800dp"
                android:background="#aaffaa"
                android:gravity="center"
                android:text="world"
                android:textColor="#000000"
                android:textSize="17sp" />
        </LinearLayout>
    </android.support.v4.widget.NestedScrollView>
</android.support.design.widget.CoordinatorLayout>
```

与上述布局文件对应的 Kotlin 页面代码如下所示：

```
class AppbarNestedActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_appbar_nested)
        setSupportActionBar(tl_title)
    }
}
```

应用栏布局配合嵌套滚动视图的演示效果如图 7-20～图 7-22 所示，其中图 7-20 展示打开演示页的初始界面，此时工具栏位于页面顶部；图 7-21 展示上拉一小段时的界面，此时工具栏随着向上滚动一段；图 7-22 展示上拉一大段时的界面，此时工具栏滚动到屏幕之外，完全看不见了。

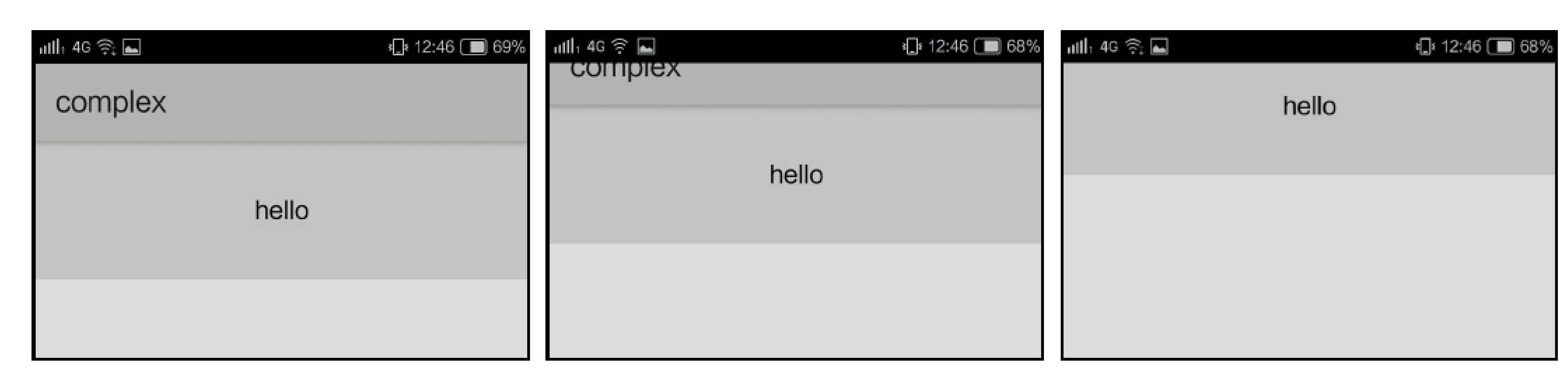


图 7-20　应用栏搭配嵌套滚动视图的初始演示界面　图 7-21　应用栏上拉一小段时的嵌套滚动视图界面　图 7-22　应用栏上拉一大段时的嵌套滚动视图界面

7.2.4　可折叠工具栏布局 CollapsingToolbarLayout

7.2.3 小节阐述了如何把 Toolbar 往上滚动，那反过来，能不能把 Toolbar 往下拉动呢？这里要明确一点，Toolbar 本身是页面顶部的工具栏，其上没有当前页面的其他控件。假如 Toolbar 拉下来，那 Toolbar 上面的空白该显示什么呢？所以 Toolbar 的上部边缘是不可以往下拉的，只有下部边缘才能往下拉，这样的视觉效果好比 Toolbar 如电影幕布一般缓缓向下展开。

不过，Android 在实现导航栏展开效果的时候，并非直接让 Toolbar 展开或收缩，而是另外提供了可折叠工具栏布局 CollapsingToolbarLayout，通过该布局节点包裹 Toolbar 节点，从而控制导航栏的展开和收缩行为。

若要在 App 工程中使用 CollapsingToolbarLayout，则需注意以下几点修改：

（1）在 build.gradle 中添加几个库的编译支持，包括 appcompat-v7 库（Toolbar 需要）、design 库（CollapsingToolbarLayout 需要）、recyclerview 库（主页面的 RecyclerView 需要）。

（2）布局文件的根布局采用 CoordinatorLayout，因为 design 库的动态效果都依赖于该控件，并且该节点要添加命名空间声明 xmlns:app="http://schemas.android.com/apk/res-auto"。

（3）使用 AppBarLayout 节点包裹 CollapsingToolbarLayout 节点，再在 CollapsingToolbarLayout 节点下添加 Toobar 节点。

（4）给 Toobar 节点添加滚动属性 app:layout_scrollFlags="scroll|enterAlways"，声明工具栏的滚动行为标志。

（5）演示界面的页面主体使用 RecyclerView 控件或者 NestedScrollView 控件，并给该控件节点添加行为属性，即 app:layout_behavior="@string/appbar_scrolling_view_behavior"，表示通知 AppBarLayout 捕捉 RecyclerView 的滚动操作。

App 在运行的时候，Toolbar 的高度是固定不变的，会发生高度变化的布局其实是 CollapsingToolbarLayout。只是许多 App 把这两者的背景设为一种颜色，所以看起来像是统一的标题栏在收缩和展开。既然二者原本不是一家，那么就得有新的属性用于区分它们内部的行为，新属性有两个，分别说明如下：

（1）折叠模式属性。属性名为 app:layout_collapseMode，它指定子视图（通常是 Toolbar）的折叠模式，折叠模式的取值说明见表 7-2。

表 7-2　折叠模式的取值说明

折叠模式取值	说明
pin	固定模式。Toolbar 固定不动，不受 CollapsingToolbarLayout 的折叠影响
parallax	视差模式。随着 CollapsingToolbarLayout 的收缩与展开，Toolbar 也跟着收缩与展开。折叠系数可通过属性 app:layout_collapseParallaxMultiplier 配置，该属性为 1.0 时，折叠效果同 pin 模式，即固定不动；该属性为 0.0 时，折叠效果等同于 none 模式，即也跟着移动相同距离
none	默认值。CollapsingToolbarLayout 折叠多少距离，Toolbar 也随着移动多少距离，通俗地说，就是夫唱妇随

（2）折叠距离系数属性。属性名为 app:layout_collapseParallaxMultiplier，它指定视差模式时的折叠距离系数，取值在 0.0～1.0 之间。若不明确指定，则该属性值默认为 0.5。

为了区分这几种折叠模式之间的差异，下面演示一个 pin 固定模式使用的布局文件例子：

```
    <android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:id="@+id/cl_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <android.support.design.widget.AppBarLayout
            android:id="@+id/abl_title"
            android:layout_width="match_parent"
            android:layout_height="160dp"
            android:background="@color/blue_light" >

            <android.support.design.widget.CollapsingToolbarLayout
                android:id="@+id/ctl_title"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                app:layout_scrollFlags="scroll|exitUntilCollapsed"
                app:contentScrim="?attr/colorPrimary"
                app:expandedTitleMarginStart="40dp" >

                <android.support.v7.widget.Toolbar
                    android:id="@+id/tl_title"
                    android:layout_width="match_parent"
                    android:layout_height="?attr/actionBarSize"
                    android:background="@color/red"
                    app:layout_collapseMode="pin" />
            </android.support.design.widget.CollapsingToolbarLayout>
        </android.support.design.widget.AppBarLayout>

        <android.support.v7.widget.RecyclerView
```

```
        android:id="@+id/rv_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />
</android.support.design.widget.CoordinatorLayout>
```

与上述布局文件对应的 Kotlin 页面代码如下所示：

```
class CollapsePinActivity : AppCompatActivity() {
    private val years = arrayOf("鼠年", "牛年", "虎年", "兔年", "龙年", "蛇年", "
马年", "羊年", "猴年", "鸡年", "狗年", "猪年")

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_collapse_pin)
        tl_title.setBackgroundColor(Color.RED)
        setSupportActionBar(tl_title)
        ctl_title.title = getString(R.string.toolbar_name)
        rv_main.layoutManager = LinearLayoutManager(this)
        rv_main.adapter = RecyclerCollapseAdapter(this, years)
    }
}
```

采取 pin 固定模式的导航栏变化效果如图 7-23～图 7-25 所示，其中图 7-23 展示刚打开页面时的初始界面，此时导航栏完全展开；图 7-24 展示往上拉动一小段之后的界面，此时导航栏下半部分向上收缩，标题文字随之上移，而上半部分红色的 Toolbar 保持不变；图 7-25 展示往上拉动一大段之后的界面，此时导航栏下半部分完全消失，标题文字全部移入上半部分红色的 Toolbar。



图 7-23　固定模式下的导航栏初始界面



图 7-24　上拉一小段时的导航栏界面



图 7-25　上拉一大段时的导航栏界面

接下来继续演示 parallax 视差模式，只要把原布局文件中的 Toolbar 节点替换为下面的内容即可，其他布局与 Kotlin 代码均保持不变：

```
            <android.support.v7.widget.Toolbar
                android:id="@+id/tl_title"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
```

```
                android:background="@color/red"
                app:layout_collapseMode="parallax"
                app:layout_collapseParallaxMultiplier="0.1" />
```

采取 parallax 视差模式的导航栏变化效果如图 7-26～图 7-28 所示，其中图 7-26 展示刚打开页面时的初始界面，此时导航栏完全展开；图 7-27 展示往上拉动一小段之后的界面，此时导航栏下半部分向上收缩，标题文字随之上移，且上半部分红色的 Toolbar 也按照比例向上收缩；图 7-28 展示往上拉动一大段之后的界面，此时导航栏下半部分完全消失，标题文字全部移入顶部，且上半部分红色的 Toolbar 也从屏幕上消失。

注意到前面几个布局文件都用到了 app:layout_scrollFlags 属性，并且有时候取值为"scroll|enterAlways"，有时候取值为"scroll|exitUntilCollapsed"，这是为什么呢？其实这个滚动标志属性来自于 AppBarLayout，它用来定义下级控件的具体滚动行为，比如先滚还是后滚、滚一半还是全部滚、自动滚还是手动滚等。



图 7-26 视差模式下的导航栏初始界面



图 7-27 上拉一小段时的导航栏界面



图 7-28 上拉一大段时的导航栏界面

首先得弄清楚为什么 AppBarLayout 划分了这几种滚动行为，所谓知其然还要知其所以然，才更有利于记忆和理解。下面是可能产生不同滚动行为的几种场景：

（1）AppBarLayout 的滚动依赖于页面主体的滚动，与页面主体相对应，可将 AppBarLayout 称作页面头部。既然一个页面分为头部和主体两部分，那么就存在谁先滚谁后滚的问题。

（2）AppBarLayout 内部的高度也可能变化，比如它嵌套了可折叠工具栏布局 CollapsingToolbarLayout。既然 AppBarLayout 的高度是变化的，那也得区分是滚一半还是滚全部。

（3）AppBarLayout 被拉动了一段还没拉完，此时一旦松开手指，一般是就地停住。但半路刹车有碍观瞻，那么就得判断是继续停着不动，还是继续向上收缩，或者继续向下展开。

上面区分好了各种滚动行为的起因与目的，再来谈谈 layout_scrollFlags 的取值情况，这个滚动标志的取值说明见表 7-3。

表 7-3 滚动标志的取值说明

滚动标志取值	说明
scroll	头部与主体一起滚动
enterAlways	头部与主体先一起滚动，头部滚到位后，主体继续向上或者向下滚。该标志需要与 scroll 同时声明

（续表）

滚动标志取值	说明
exitUntilCollapsed	该标志保证页面上至少能看到最小化的工具栏，不会完全看不到工具栏。该标志需要与 scroll 同时声明
enterAlwaysCollapsed	该标志与 enterAlways 的区别在于有折叠操作，而单独的 enterAlways 没有折叠。该标志需要与 scroll、enterAlways 同时声明
snap	在用户手指松开时，系统自行判断，接下来是全部向上滚到顶，还是全部向下展开。该标志需要与 scroll 同时声明

7.2.5　仿支付宝首页的头部伸缩特效

前面几个小节介绍了与顶部导航栏相关的几个布局用法，可要是在实战中派不上用场，又有什么用？确实，如果一项技术没有真正用起来，那就只是花拳绣腿而已。但是协调布局等控件绝非等闲之辈，既然它们由 Android 在 MaterialDesign 库中隆重推出，肯定有大用途。下面来看两张导航栏的效果图，如图 7-29 所示，这是某 App 导航栏完全展开时的界面，此时页面头部的导航栏占据较大部分的高度；如图 7-30 所示，这是该 App 导航栏完全收缩时的界面，此时头部导航栏只剩矮矮的一个长条。





图 7-29　仿支付宝首页的头部展开效果



图 7-30　仿支付宝首页的头部缩起效果

看起来很眼熟是不是，这两张界面截图分明很像支付宝首页的头部效果。如果读者已经熟悉运用 AppBarLayout 和 CollapsingToolbarLayout，也许很快便可以做出类似以上效果的简单界面。概括地说，就是定义一个协调布局 CoordinatorLayout，然后嵌套应用栏布局 AppBarLayout，再嵌套可折叠工具栏布局 CollapsingToolbarLayout，再嵌套工具栏 Toolbar 的页面布局。支付宝首页之所以要嵌套这么多层，是因为要完成以下功能：

（1）CoordinatorLayout 嵌套 AppBarLayout，这是为了让头部导航栏能够跟随视图主体下拉而展开，并且跟随视图主体上拉而收缩。这个视图主体可以是 RecyclerView，也可以是 NestedScrollView。

（2）AppBarLayout 嵌套 CollapsingToolbarLayout，这是为了定义导航栏下面需要展开和收缩的这部分视图。

（3）CollapsingToolbarLayout 嵌套 Toolbar，这是为了声明导航栏上方无论何时都要显示的长条区域，其中 Toolbar 还要定义两个不同的下级布局，用于分别显示展开与收缩两种状态时的工具栏界面。

下面是基于以上思路实现的仿支付宝首页布局文件例子：

```
    <android.support.design.widget.CoordinatorLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fitsSystemWindows="true" >

        <android.support.design.widget.AppBarLayout
            android:id="@+id/abl_bar"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:fitsSystemWindows="true" >

            <android.support.design.widget.CollapsingToolbarLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:fitsSystemWindows="true"
                app:layout_scrollFlags="scroll|exitUntilCollapsed|snap"
                app:contentScrim="@color/blue_dark" >

                <!-- life_pay.xml 定义了工具栏下方的频道布局 -->
                <include
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:layout_marginTop="@dimen/toolbar_height"
                    app:layout_collapseMode="parallax"
                    app:layout_collapseParallaxMultiplier="0.7"
                    layout="@layout/life_pay" />

                <android.support.v7.widget.Toolbar
                    android:layout_width="match_parent"
                    android:layout_height="@dimen/toolbar_height"
                    app:layout_collapseMode="pin"
                    app:contentInsetLeft="0dp"
                    app:contentInsetStart="0dp" >

                    <!-- toolbar_expand.xml 定义了展开状态时的工具栏内容布局 -->
                    <include
                        android:id="@+id/tl_expand"
```

```
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                layout="@layout/toolbar_expand" />

            <!-- toolbar_collapse.xml 定义了收缩状态时的工具栏内容布局 -->
            <include
                android:id="@+id/tl_collapse"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                layout="@layout/toolbar_collapse"
                android:visibility="gone" />
        </android.support.v7.widget.Toolbar>
    </android.support.design.widget.CollapsingToolbarLayout>
</android.support.design.widget.AppBarLayout>

<android.support.v7.widget.RecyclerView
    android:id="@+id/rv_content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="10dp"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
</android.support.design.widget.CoordinatorLayout>
```

然而仅实现上述布局并非万事大吉，因为支付宝首页的头部在伸缩时可是有动画效果的，就像图 7-31 和图 7-32 所示的淡入淡出渐变动画。



图 7-31　头部导航栏的淡入效果



图 7-32　头部导航栏的淡出效果

图 7-31 和图 7-32 所体现的渐变动画其实可分为两部分：

（1）导航栏从展开状态向上收缩时，头部的各个控件要慢慢向背景色过渡，也就是淡入效果。

（2）导航栏向上收缩到一半，顶部的工具栏要更换成收缩状态下的工具栏布局，并且随着导航栏继续向上收缩，新工具栏上的各个控件也要慢慢变得清晰起来，也就是淡出效果。

若导航栏是从收缩状态向下展开的，则此时相应地做上述渐变动画的取反效果，即下面的描述：

（1）导航栏从收缩状态向下展开时，头部的各个控件要慢慢向背景色过渡，也就是淡入效果；同时展开导航栏的下部分布局，并且该布局上的各个控件渐渐变得清晰。

（2）导航栏向下展开到一半，顶部的工具栏要更换成展开状态下的工具栏布局，并且随着导航栏继续向下展开，新工具栏上的各个控件也要慢慢变得清晰起来，也就是淡出效果。

看文字描述还比较复杂，如果只对某个控件做渐变动画还好，可是导航栏上的控件有好几个，而且数量并不固定，常常会增加新控件或者修改原控件。倘若要对导航栏上的各个控件逐一展示动画，不但花费力气，而且后期也不好维护。为了解决这个动画问题，可以采取类似遮罩的做法，即一开始先给导航栏罩上一层透明的视图，此时导航栏的界面就完全显示；然后随着导航栏的移动距离计算当前位置下的遮罩透明度，使该遮罩变得越来越不透明，看起来导航栏像是蒙上了一层薄雾面纱，蒙到最后就完全看不见了。反过来，也可以一开始给导航栏罩上一层不透明的视图，此时导航栏的所有控件都是看不见的，然后随着距离的变化，遮罩变得越来越不透明，导航栏也会跟着变得越来越清晰。

现在渐变动画的思路有了，可谓万事俱备，只欠东风，再注册一个导航栏的位置偏移监听事件便行，正好有个现成的监听器 AppBarLayout.OnOffsetChangedListener，只需给应用栏布局对象调用 addOnOffsetChangedListener 方法，即可实现给导航栏注册偏移监听器的功能。接下来看下面具体的 Kotlin 实现代码：

```
    //因为布局文件通过 include 加载了多个子布局，所以 Kotlin 代码也要同时 import 导入所有相关
的布局
    import kotlinx.android.synthetic.main.activity_scroll_alipay.*
    import kotlinx.android.synthetic.main.life_pay.*
    import kotlinx.android.synthetic.main.toolbar_expand.*
    import kotlinx.android.synthetic.main.toolbar_collapse.*

    class ScrollAlipayActivity : AppCompatActivity(), OnOffsetChangedListener {
        private var mMaskColor: Int = 0

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_scroll_alipay)
            mMaskColor = resources.getColor(R.color.blue_dark)
            //给控件 abl_bar 注册一个位置偏移的监听器
            abl_bar.addOnOffsetChangedListener(this)
            rv_content.layoutManager = GridLayoutManager(this, 4)
            //第一种方式：使用采取了 LayoutContainer 的适配器
            //rv_content.adapter = RecyclerLifeAdapter(this, LifeItem.default)
            //第二种方式：使用把三类可变要素抽象出来的适配器
            rv_content.adapter = RecyclerCommonAdapter(this, R.layout.item_life,
LifeItem.default,
                    { view, item ->
                        val iv_pic = view.findViewById<ImageView>(R.id.iv_pic)
                        val tv_title = view.findViewById<TextView>(R.id.tv_title)
```

```
                    iv_pic.setImageResource(item.pic_id)
                    tv_title.text = item.title
                })
        }

        override fun onOffsetChanged(appBarLayout: AppBarLayout, verticalOffset:
Int) {
            val offset = Math.abs(verticalOffset)
            val total = appBarLayout.totalScrollRange
            val alphaOut = if (200 - offset < 0) 0 else 200 - offset
            //计算淡入时候的遮罩透明度
            val maskColorIn = Color.argb(offset, Color.red(mMaskColor),
                    Color.green(mMaskColor), Color.blue(mMaskColor))
            //工具栏下方的频道布局要加速淡入或者淡出
            val maskColorInDouble = Color.argb(offset * 2, Color.red(mMaskColor),
                    Color.green(mMaskColor), Color.blue(mMaskColor))
            //计算淡出时候的遮罩透明度
            val maskColorOut = Color.argb(alphaOut * 3, Color.red(mMaskColor),
                    Color.green(mMaskColor), Color.blue(mMaskColor))
            if (offset <= total / 2) { //若偏移量小于一半，则显示展开时候的工具栏
                tl_expand.visibility = View.VISIBLE
                tl_collapse.visibility = View.GONE
                v_expand_mask.setBackgroundColor(maskColorInDouble)
            } else { //若偏移量大于一半，则显示缩小时候的工具栏
                tl_expand.visibility = View.GONE
                tl_collapse.visibility = View.VISIBLE
                v_collapse_mask.setBackgroundColor(maskColorOut)
            }
            v_pay_mask.setBackgroundColor(maskColorIn)
        }
    }
```

本节至此基本过了一遍 MaterialDesign 库的主要控件用法，虽然没涉及什么新的 Kotlin 语法知识，但是复习现有的语法也不错，温故而知新，慢慢来。

7.3　实现页面切换

前两节介绍的视图排列和协调布局基本上属于上下滚动，再深入也只是让上下滚动的花样变得丰富一些。可是许多 App 除了上下滚动操作外，还有左右滑动的页面切换操作，那么页面的左右滑动又是怎样实现的呢？本节就对页面的左右滑动展开说明，从翻页视图到碎片适配再到标签布局，逐步介绍页面切换的几种实现方式。

7.3.1 翻页视图 ViewPager

由于手机屏幕是竖长形状的，高度比宽度大，因此页面在多数情况下是上下滚动的。然而上下滚动仅限于一维方向，无法有效利用屏幕空间，所以往往还需要支持左右滑动，通过手势的左滑或者右滑来模拟现实生活中的翻页效果，于是便有了翻页视图 ViewPager。

对于翻页视图来说，一个页面就是一个单项（相当于 ListView 的一个列表项），许多个页面组成 ViewPager 的页面项。已经明确了 ViewPager 的原理类似 ListView 和 GridView，ViewPager 的用法也与它们类似。列表视图和网格视图的适配器使用基本适配器 BaseAdapter，翻页视图的适配器则用翻页适配器 PagerAdapter；列表视图和网格视图的监听器使用 OnItemClickListener，翻页视图的监听器则用 OnPageChangeListener，表示监听页面切换事件。表 7-4 给出了翻页视图相关方法/属性的 Kotlin 与 Java 方法调用方式的对比关系。

表 7-4　翻页视图相关方法/属性的 Kotlin 与 Java 方法调用方式对比

方法/属性说明	Kotlin 的调用方式	Java 的调用方式
设置页面项的适配器	adapter	setAdapter
设置当前的页码	currentItem	setCurrentItem
设置翻页视图的页面切换监听器	addOnPageChangeListener	addOnPageChangeListener

翻页适配器 PagerAdapter 与基本适配器 BaseAdapter 的用法相近，也需实现构造函数，获取页面个数的 getCount 方法，生成单个页面视图的 instantiateItem 方法，另外多了一个回收页面的 destroyItem 方法。举个左右翻动手机图片进行浏览的例子，每个页面都是单独的图像视图 ImageView，所有图像页面通过翻页适配器组合到翻页视图之中。下面是使用 PagerAdapter 实现浏览图片效果的 Kotlin 代码示例：

```
    //在主构造函数中声明与入参同名的属性及其初始赋值操作
    class ImagePagerAdapater(private val context: Context, private val goodsList: MutableList<GoodsInfo>) : PagerAdapter() {
        private val views = mutableListOf<ImageView>()

        //初始化函数进行开发者额外的初始操作
        init {
            for (item in goodsList) {
                val view = ImageView(context)
                view.layoutParams = LayoutParams(LayoutParams.MATCH_PARENT, LayoutParams.WRAP_CONTENT)
                view.setImageResource(item.pic)
                view.scaleType = ScaleType.FIT_CENTER
                views.add(view)
            }
        }

        //获取页面数量，使用了简化函数
```

```
        override fun getCount(): Int = views.size

        //判断指定页面是否已加入适配器，注意这里用到了引用相等
        override fun isViewFromObject(arg0: View, arg1: Any): Boolean = (arg0 ===
arg1)

        //回收页面
        override fun destroyItem(container: ViewGroup, position: Int, `object`: Any) {
            container.removeView(views[position])
        }

        //实例化每个页面
        override fun instantiateItem(container: ViewGroup, position: Int): Any {
            container.addView(views[position])
            return views[position]
        }

        //获得页面的标题，要跟 PagerTabStrip 配合使用
        override fun getPageTitle(position: Int): CharSequence =
goodsList[position].name
    }
```

与上面适配器对应的 Kotlin 页面代码如下所示：

```
    class ViewPagerActivity : AppCompatActivity(), OnPageChangeListener {
        private var goodsList = GoodsInfo.defaultList

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_view_pager)
            //注意 PagerTabStrip 不存在 textSize 属性，只能调用 setTextSize 方法设置文字大小
            pts_tab.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20f)
            pts_tab.setTextColor(Color.GREEN)
            vp_content.adapter = ImagePagerAdapater(this, goodsList)
            vp_content.currentItem = 0
            vp_content.addOnPageChangeListener(this)
        }

        override fun onPageScrollStateChanged(arg0: Int) {}

        override fun onPageScrolled(arg0: Int, arg1: Float, arg2: Int) {}

        //在页面切换结束（即滑动停止）时触发该方法
        override fun onPageSelected(arg0: Int) {
            toast("您翻到的手机品牌是：${goodsList[arg0].name}")
        }
    }
```

从以上代码看到，使用 Kotlin 书写的页面代码稀松平常，倒是翻页适配器 ImagePagerAdapater 的 Kotlin 实现代码大有讲究，主要有以下几个要点：

（1）主构造函数只能自动完成与入参同名的属性声明及其初始赋值操作，如果还存在其他初始化操作，就需要在 init 函数中完成。

（2）isViewFromObject 方法判断指定页面是否已加入适配器，因为是判断唯一性，所以为了防止盗版，必须通过引用相等来校验。

（3）getPageTitle 方法必须配合翻页标签栏 PagerTabStrip 或者翻页标题栏 PagerTitleStrip 控件才能正常显示页面上方的标题文字。

由此可见，翻页适配器 Kotlin 代码的关键之处是引用相等。分析代码完毕，接着观察一下翻页视图的滚动效果，如图 7-33 所示，在截图的瞬间，ViewPager 正巧在左右两个页面之间滑动。

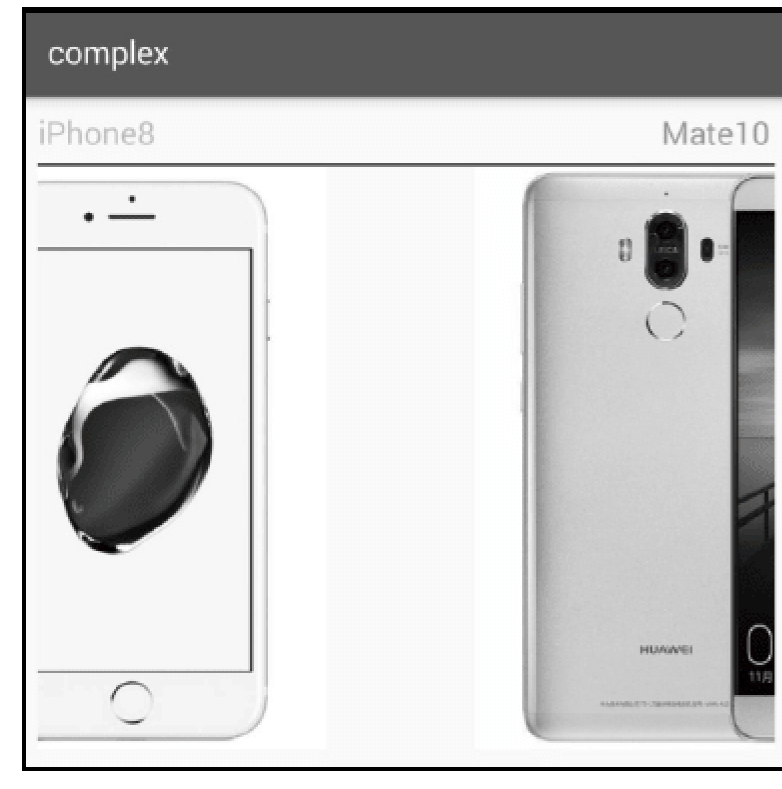


图 7-33　翻页视图的滚动瞬间界面

7.3.2　碎片 Fragment

顾名思义，碎片 Fragment 只是页面的片段，并非完整的页面，故而 Fragment 很少单独使用，基本都要跟其他控件配合。最经常跟 Fragment 搭档的好伙伴自然是 ViewPager，不过普通的翻页适配器 PagerAdapter 没法满足 Fragment，必须搭配 Fragment 的好基友碎片适配器 FragmentStatePagerAdapter 才行。

接下来，使用 Kotlin 编码将 ViewPager+Fragment 的翻页全流程实现出来，首先要定义每个页面的动态碎片，依旧以手机商品为例，演示用的 Kotlin 碎片类代码如下：

```
class DynamicFragment : Fragment() {
    private var ctx: Context? = null
    private var mPosition: Int = 0
    private var mImageId: Int = 0
    private var mDesc: String? = null
    private var mPrice: Int = 0

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?): View {
        ctx = activity
        //碎片内部通过 arguments 获取外部的输入参数
        if (arguments != null) {
            mPosition = arguments!!.getInt("position", 0)
            mImageId = arguments!!.getInt("image_id", 0)
            mDesc = arguments!!.getString("desc")
            mPrice = arguments!!.getInt("price")
        }
        val view = inflater.inflate(R.layout.fragment_dynamic, container,
false)
        //注意 Fragment 内部仍需通过 findViewById 获得控件对象
```

```
            val iv_pic = view.findViewById<ImageView>(R.id.iv_pic)
            val tv_desc = view.findViewById<TextView>(R.id.tv_desc)
            iv_pic.setImageResource(mImageId)
            tv_desc.text = "$mDesc\n 售价：$mPrice"
            return view
        }

        companion object {
            //利用伴生对象定义获取碎片实例的静态方法
            fun newInstance(position: Int, image_id: Int, desc: String, price: Int):
DynamicFragment {
                val fragment = DynamicFragment()
                val bundle = Bundle()
                bundle.putInt("position", position)
                bundle.putInt("image_id", image_id)
                bundle.putString("desc", desc)
                bundle.putInt("price", price)
                //外部通过 arguments 向碎片传递输入参数
                fragment.arguments = bundle
                return fragment
            }
        }
    }
```

正如上面代码中注释描述的那样，使用 Kotlin 书写碎片类主要有三个需要注意的地方：

（1）在 Fragment 类中获取控件对象依然要调用 findViewById 方法。对比之下，Activity 类中早已支持直接操作控件对象，即使是循环视图的适配器，也能通过插件 LayoutContainer 来自动获得控件对象。可是经常用到的 Fragment 类尚未得到优化，只能期待将来 Kotlin 补充这方面的支持了。

（2）碎片类采用 Java 编码时，通常会提供一个静态方法 newInstance，提供给外部以获取该碎片的实例。而在 Kotlin 中，若要实现静态方法，则需借助于伴生对象。

（3）外部向碎片传递信息，数据中转站是 Fragment 对象的情景参数 arguments 属性，而非 Activity 之间传递信息用到的 intent 属性。其实 arguments 与 intent 的功能类似，都是用于组件之间传递消息，只是前者为 Activity 向 Fragment 传数据，而后者为 Activity 向另一个 Activity 传数据。

分析完了碎片类的 Kotlin 实现代码，再来看看手机商品例子的碎片适配器，具体的 Kotlin 代码如下所示：

```
    class MobilePagerAdapter(fm: FragmentManager, private val goodsList:
MutableList<GoodsInfo>) : FragmentStatePagerAdapter(fm) {

        //获取页面的数量
        override fun getCount(): Int = goodsList.size

        //获取每个页面的碎片对象
        override fun getItem(position: Int): Fragment {
            val item = goodsList[position]
```

```
            return DynamicFragment.newInstance(position, item.pic, item.desc,
item.price)
        }

        //获取页面的标题
        override fun getPageTitle(position: Int): CharSequence =
goodsList[position].name
    }
```

可见上述的碎片适配器代码实在精简，比起对应的 Java 代码来又瘦身了不少。特别注意，碎片适配器定义了一个 FragmentManager 对象的输入参数，该入参需要由 Activity 页面在调用时填写，下面是详细的 Kotlin 页面代码：

```
    class FragmentDynamicActivity : FragmentActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_fragment_dynamic)
            pts_tab.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20f)
            //碎片适配器需要传入碎片管理器对象 supportFragmentManager
            vp_content.adapter = MobilePagerAdapter(supportFragmentManager,
GoodsInfo.defaultList)
            vp_content.currentItem = 0
        }
    }
```

依例再看看碎片结合翻页视图的展示效果，如图 7-34 所示，此时向右翻到华为手机的界面；如图 7-35 所示，此时继续向右翻到 OPPO 手机的界面。



图 7-34 翻到华为手机的翻页视图界面



图 7-35 翻到 OPPO 手机的翻页视图界面

7.3.3　标签布局 TabLayout

前面两个小节介绍了翻页视图的两种使用方式：普通视图与碎片布局，不知读者有没有注意到，这两种方式呈现出来的界面效果，在视图主体上方会有两排控件。如图 7-36 所示，最上面一排为系统自带的导航栏，导航栏下面一排为 ViewPager 的标题搭档 PagerTabStrip。



图 7-36　同时存在导航栏和标签页的翻页视图界面

然而手机屏幕的空间本来就有限，现在却有两排都属于导航功能的控件，实在是挥霍宝贵的显示区域。既然这两排同样具备导航功能，不如就此将它们合二为一，形成统一的顶部导航，如此岂不更好？这个思路提到的顶部导航栏正可以使用上一节提到的工具栏 Toolbar，通过 Toolbar 定制开发者想要的导航栏目。至于 PagerTabStrip 所展现的文本标签切换，则可利用 MaterialDesign 库里的新控件——标签布局 TabLayout，把标签布局结合翻页视图使用，就能在视觉上再现 ViewPager 加 PagerTabStrip 的翻页标签切换效果。

因为标签布局 TabLayout 来自于 MaterialDesign 库，所以使用该控件前要先修改 build.gradle，在 dependencies 节点中加入下面一行表示导入 design 库：

```
//需要将“$supportVersion”替换为读者电脑上的 design 库版本号
compile 'com.android.support:design:$supportVersion'
```

标签布局的展现形式类似 PagerTabStrip，一样是文字标签带下划线。二者之间不同的是，TabLayout 允许定制更丰富的样式，它新增的样式属性主要有以下几种。

- tabBackground：指定标签的背景。
- tabIndicatorColor：指定下划线的颜色。
- tabIndicatorHeight：指定下划线的高度。
- tabTextColor：指定标签文字的颜色。
- tabTextAppearance：指定标签文字的风格。
- tabSelectedTextColor：指定选中文字的颜色。

在代码中，TabLayout 通过如下方法操作标签元素。

- newTab：创建新标签。
- addTab：添加一个标签。
- getTabAt：获取指定位置的标签。
- setOnTabSelectedListener：设置标签的选中监听器。

接下来要实现统一的顶部导航栏，得在布局文件中由 Toolbar 节点包裹 TabLayout 节点，表示在工具栏框架中添加标签布局控件。具体的布局文件例子如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <android.support.v7.widget.Toolbar
        android:id="@+id/tl_head"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        app:navigationIcon="@drawable/ic_back" >

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

            <android.support.design.widget.TabLayout
                android:id="@+id/tab_title"
                android:layout_width="wrap_content"
                android:layout_height="match_parent"
                android:layout_centerInParent="true"
                app:tabIndicatorColor="@color/red"
                app:tabIndicatorHeight="2dp"
                app:tabSelectedTextColor="@color/red"
                app:tabTextColor="@color/grey"
                app:tabTextAppearance="@style/TabText" />

        </RelativeLayout>
    </android.support.v7.widget.Toolbar>

    <!-- 这是一条顶部导航栏与页面主体的分隔线 -->
    <View
        android:layout_width="match_parent"
        android:layout_height="1dp"
        android:background="@color/grey" />

    <android.support.v4.view.ViewPager
        android:id="@+id/vp_content"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

那么与上面布局文件对应的 Kotlin 代码需要把翻页视图与标签布局二者关联起来。关联的方式是通过监听器联动，翻页视图利用页面切换监听器 SimpleOnPageChangeListener 去同步标签布局，而标签布局利用标签选中监听器 OnTabSelectedListener 去同步翻页视图。下面是商品翻页信息的 Kotlin 页面代码例子：

```
class TabLayoutActivity : AppCompatActivity(), OnTabSelectedListener {
    private val titles = mutableListOf<String>("商品", "详情")

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_tab_layout)
        //使用自定义的工具栏替换系统默认的导航栏
        setSupportActionBar(tl_head)
        initTabLayout()
        initTabViewPager()
    }

    //初始化头部的文本标签
    private fun initTabLayout() {
        tab_title.addTab(tab_title.newTab().setText(titles[0]))
        tab_title.addTab(tab_title.newTab().setText(titles[1]))
        tab_title.addOnTabSelectedListener(this)
    }

    //初始化页面主体的翻页视图
    private fun initTabViewPager() {
        vp_content.adapter = GoodsPagerAdapter(supportFragmentManager,
titles)
        //利用 object 关键字表示声明一个匿名实例
        vp_content.addOnPageChangeListener(object :
SimpleOnPageChangeListener() {
            override fun onPageSelected(position: Int) {
                //翻页操作停止后，同步切换到对应的文本标签
                tab_title.getTabAt(position)!!.select()
            }
        })
    }

    override fun onTabReselected(tab: Tab) {}

    //文本标签选中后，同步切换到对应的翻页页面
    override fun onTabSelected(tab: Tab) {
        vp_content.currentItem = tab.position
    }

    override fun onTabUnselected(tab: Tab) {}
}
```

翻页视图内部还需通过碎片适配器加载具体的每个碎片页面，以下是对应的 Kotlin 碎片适配器代码，仅简单加载了两个碎片页 BookCoverFragment 和 BookDetailFragment：

```
    class GoodsPagerAdapter(fm: FragmentManager, private val titles:
MutableList<String>) : FragmentPagerAdapter(fm) {

        //根据位置序号分别指定不同的 Fragment 碎片类
        override fun getItem(position: Int): Fragment = when (position) {
            0 -> BookCoverFragment()
            1 -> BookDetailFragment()
            else -> BookCoverFragment()
        }

        override fun getCount(): Int = titles.size

        override fun getPageTitle(position: Int): CharSequence = titles[position]
    }
```

最终运行之后，展现了位于统一导航栏之下的翻页视图，它的翻页切换效果如图 7-37 和图 7-38 所示。其中，图 7-37 所示为展示“商品”标签页时的界面，此时导航栏提示当前为商品页；图 7-38 所示为展示“详情”标签页时的界面，此时导航栏提示当前为详情页。

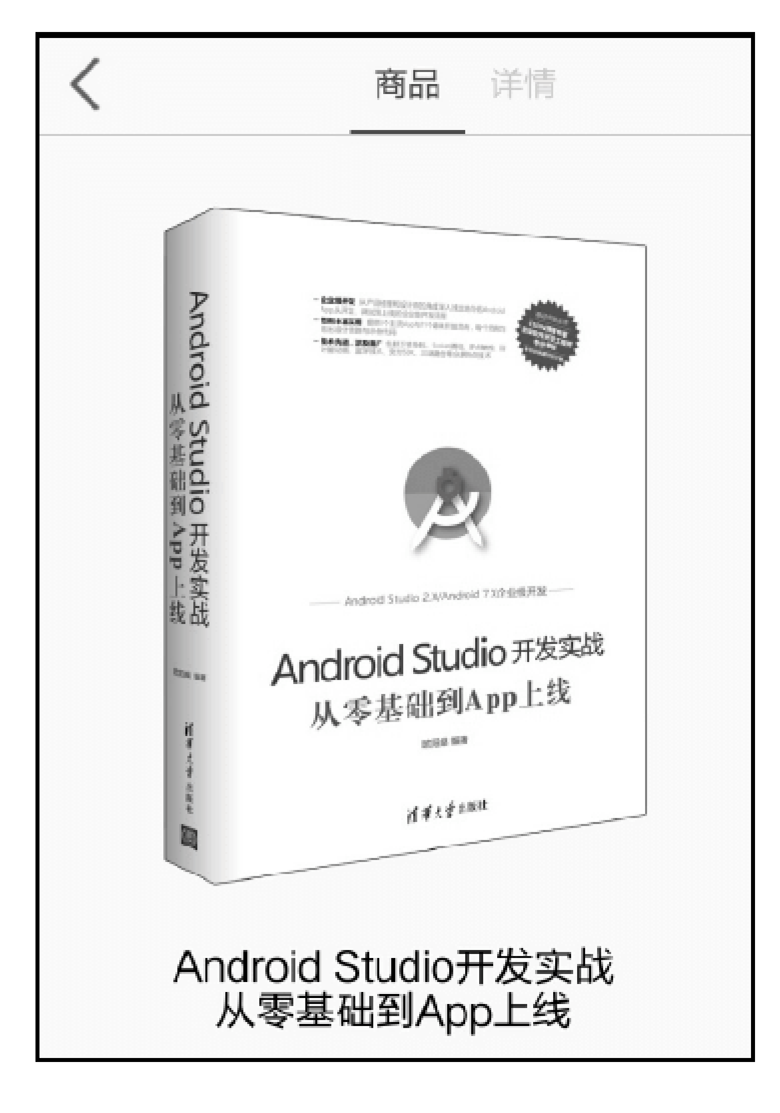


图 7-37　展示“商品”标签页的界面

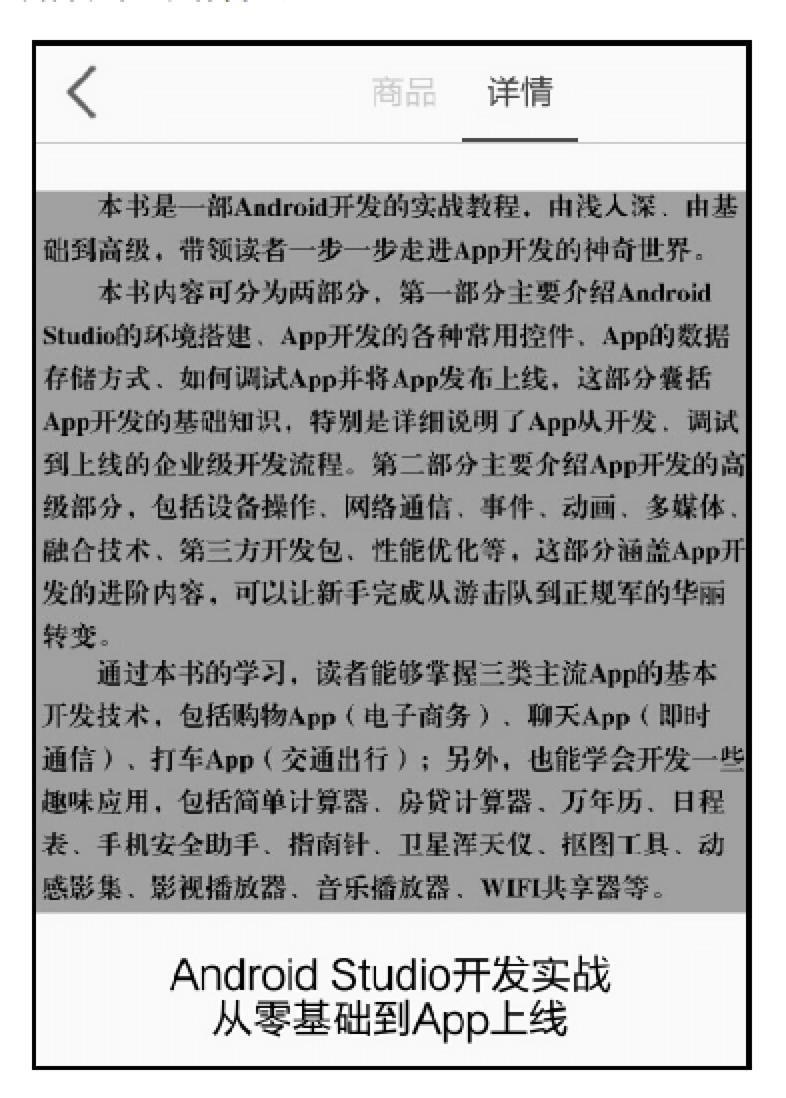


图 7-38　展示“详情”标签页的界面

7.4　广播收发 Broadcast

一个应用的功能越强大，它所包含的组件和控件的数量就越多，不断增长的组件/控件之间要想进行有效的数据传输，有赖于一种更灵活、更方便的消息通信机制。在 Android 系统中，有一种灵活通信的消息机制被塑造成四大组件之一的广播 Broadcast。本节就对广播的场景、过程、实现方式等概念以及具体用法进行详细的说明介绍。

7.4.1　收发临时广播

App 在运行过程中，各组件之间常常要进行数据交互，常见的数据交互场景主要有以下几种：

（1）一个页面 Acitivity 向另一个页面 Acitivity 传递消息，按照第 6 章的“6.4 Activity 活动跳转”一节的描述，可使用 Intent 对象传输请求信息。对应地，下一个页面向上一个页面返回消息也可利用 Intent 对象传输应答信息。

（2）页面 Acitivity 向各类适配器传递消息可通过适配器的主构造函数传入参数。反过来，适配器向页面 Acitivity 返回消息则稍微有些麻烦，这时要先定义一个监听器，由 Acitivity 向适配器传入监听器对象，然后适配器在合适的时候调用监听器的方法（Activity 要事先实现该监听器的内部方法），从而间接实现适配器回调页面的要求。

（3）页面 Acitivity 向碎片 Fragment 传递消息时，按照前面“7.3.2 碎片 Fragment”小节的叙述，可在碎片适配器中给碎片对象设置情景参数 arguments 属性，然后碎片内部从 arguments 中获取具体的请求信息。但若是碎片向页面回传消息，这个麻烦就大了，因为此时既没有 Activity 之间的 ForResult 方法，也没法传入监听器对象，那么碎片如何把消息传回给页面?

像上面的第三种数据交互情景，Android 中还有好些类似的无法直接传递数据的情况，这时候便用到了广播 Broadcast 组件。广播如其名，它发送消息时，并未指明要发给哪个特定对象，而是面向大众广而播之，故而台下的听众只要有在倾听皆可接收到广播内容。由此看来，广播特别适用于 Android 组件之间的灵活通信，它与 Activity 的区别在于下列几点：

（1）Activity 只能一对一地通信，而 Broadcast 可以一对多，一人发送广播，多人接收处理。

（2）对于发送者来说，广播不需要考虑接收者有没有在工作，接收者有在工作则接收广播，不在工作则丢弃广播。

（3）对于接收者来说，会收到各式各样的广播，所以接收者首先要自行过滤哪些是符合条件的，然后才能进行解包处理。

与广播有关的方法主要有以下三个。

- sendBroadcast：发送广播。
- registerReceiver：注册广播接收器。接收器只有在注册之后，才能正常接收广播消息。
- unregisterReceiver：注销广播接收器。

为了更好地说明广播的工作流程，接下来还是对其进行具体的演示。譬如，Fragment 内部有个下拉框，可下拉选择背景颜色，一旦选中某个背景色，则整个活动页面的背景色都换成新颜色。那么 Fragment 内部发现选中新颜色后，就要发送一个背景色变更的广播。下面是 Fragment 内部实现广播发送的 Kotlin 关键代码片段：

```
    private val colorNames = listOf("红色", "黄色", "绿色", "青色", "蓝色")
    private val colorIds = intArrayOf(Color.RED, Color.YELLOW, Color.GREEN,
Color.CYAN, Color.BLUE)
```

```
    //初始化选择背景色的下拉框
    private fun initSpinner() {
        sp_bg.visibility = View.GONE
        tv_spinner.visibility = View.VISIBLE
        tv_spinner.text = colorNames[mSeq]
        tv_spinner.setOnClickListener {
            ctx!!.selector("请选择页面背景色", colorNames) { i ->
                tv_spinner.text = colorNames[i]
                mSeq = i
                //设置广播意图的名称为 BroadcastFragment.EVENT
                val intent = Intent(BroadcastFragment.EVENT)
                intent.putExtra("seq", i)
                intent.putExtra("color", colorIds[i])
                //已选择新颜色，则发送背景色变更的广播
                ctx!!.sendBroadcast(intent)
            }
        }
    }

    companion object {
        //静态属性如果是个常量，就还要添加修饰符 const
        const val EVENT = "com.example.complex.fragment.BroadcastFragment"

        fun newInstance(position: Int, image_id: Int, desc: String):
BroadcastFragment {
            val fragment = BroadcastFragment()
            val bundle = Bundle()
            bundle.putInt("position", position)
            bundle.putInt("image_id", image_id)
            bundle.putString("desc", desc)
            fragment.arguments = bundle
            return fragment
        }
    }
```

因为广播的发送和接收不依赖于任何组件中转，所以适配器代码无须添加任何广播处理代码，只需在 Activity 页面代码中补充广播接收处理即可。添加了广播接收处理的 Kotlin 页面代码如下所示：

```
class BroadTempActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_broadcast_temp)
        pts_tab.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20f)
```

```
            vp_content.adapter = BroadcastPagerAdapter(supportFragmentManager,
GoodsInfo.defaultList)
            vp_content.currentItem = 0
        }

        public override fun onStart() {
            super.onStart()
            bgChangeReceiver = BgChangeReceiver()
            //声明一个过滤器，明确只接收名称为 BroadcastFragment.EVENT 的广播
            val filter = IntentFilter(BroadcastFragment.EVENT)
            //在活动启动时注册广播接收器
            registerReceiver(bgChangeReceiver, filter)
        }

        public override fun onStop() {
            //在活动停止时注销广播接收器
            unregisterReceiver(bgChangeReceiver)
            super.onStop()
        }

        private var bgChangeReceiver: BgChangeReceiver? = null
        //定义一个背景色变更的广播接收器
        private inner class BgChangeReceiver : BroadcastReceiver() {
            override fun onReceive(context: Context, intent: Intent?) {
                if (intent != null) {
                    //从广播消息中获取新颜色，并将页面背景色修改成新颜色
                    val color = intent.getIntExtra("color", Color.WHITE)
                    ll_brd_temp.setBackgroundColor(color)
                }
            }
        }
    }
```

上述的碎片以及页面代码新增的 Kotlin 语法主要是修饰符 const。倘若按照字面意思，被 const 修饰的属性是个常量属性，似乎跟 val 的只读变量没什么区别，那为什么代码声明 EVENT 这个广播事件名称时，既加了 const 修饰又加了 val 修饰呢？这说明二者之间其实还是有所差别的，要想弄清楚其中的差异，得先了解两种常量概念：编译时常量和运行时常量。

1. 编译时常量

这种类型的常量的值早在编译期间就已经确定，相当于这个常量值被固化到了 App 安装包里面。无论 App 在哪部手机上安装、在何时运行，编译时常量的值都是统一且唯一的，不会随环境的变化产生任何变化。

2. 运行时常量

这种类型的常量其实不是严格意义上的常量，更确切地说，应该是一个仅能赋值一次的只读

属性。运行时常量的赋值操作可以在声明属性时就赋值，也可以在首次使用时赋值，并且赋值的时候，还可以把另一个变量的数值赋给 val 变量。也就是说，App 在每次启动运行之后，运行时常量都可能被赋予不同的数值，只是一旦完成赋值，其值就不能再做修改。

由此可见，编译时常量才是真正意义上的常量，而运行时常量是容易使人迷惑的伪常量。之所以前面的 Kotlin 代码将 EVENT 声明为 const 常量，是因为系统注册广播接收器时要求这个广播的名称是唯一的，不然这次运行是这个广播名称，下次运行又是另一个广播名称，实在让系统无所适从。

解释完了 const 的用法，再来看看实际运行的广播效果，一开始页面背景是淡灰色的，如图 7-39 所示；接着在碎片内部的下拉框中选择红色，于是整个页面的背景色都变成红色了，如图 7-40 所示。



图 7-39　临时广播未发送前的界面



图 7-40　临时广播发送后的界面

7.4.2　接收系统广播

7.4.1 小节介绍的临时广播指的是 App 自身发出来的局部广播，一旦该 App 退出运行，就无法继续收发临时广播。显然这个临时广播的受众面狭小，如果 App 希望满足某种特定条件（如开机启动、用户解锁、时刻到达、网络切换等），就自动执行相应的事务处理，这要求 App 去接收来自 Android 系统的系统广播。系统广播是 Android 发现产生某种系统事件之时，向手机上所有应用发送通知的全局广播。

系统广播的注册方式有两种：静态注册和动态注册，分别说明如下。

1. 静态注册方式

静态注册适用于开机启动、用户解锁、定时闹钟等系统事件，该方式无论 App 当前是否正在运行，只要注册了系统广播的接收器，一旦广播对应的系统事件发生，那么 App 都得恢复运行去处理接收器的事务。

静态注册方式要在 AndroidManifest.xml 中添加广播接收器的 receiver 节点配置，详细的配置方法示例如下：

```
<receiver android:name=".receiver.BootCompletedReceiver" >
    <intent-filter>
```

```
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>
```

然后编写 BootCompletedReceiver 类的具体代码，在类内部进行响应开机启动的业务逻辑操作。

2. 动态注册方式

动态注册适用于分钟到达广播、网络切换广播、电量变化广播等，由于该方式必须在 App 代码中注册广播接收器，因此只有 App 启动之后才能正常接收广播。假如 App 没有启动，或者启动之后又退出运行，那就不能再接收广播了。

下面是采取动态注册方式实现监听分钟广播的 Kotlin 代码例子：

```
class BroadSystemActivity : AppCompatActivity() {
    var desc = "开始侦听分钟广播，请稍等。注意要保持屏幕亮着，才能正常收到广播"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_broadcast_system)
        tv_system.text = desc
    }

    override fun onStart() {
        super.onStart()
        timeReceiver = TimeReceiver()
        //声明一个过滤器，只接收名称为 Intent.ACTION_TIME_TICK 的分钟广播
        val filter = IntentFilter(Intent.ACTION_TIME_TICK)
        //在活动启动时注册广播接收器
        registerReceiver(timeReceiver, filter)
    }

    override fun onStop() {
        super.onStop()
        //在活动停止时注销广播接收器
        unregisterReceiver(timeReceiver)
    }

private var timeReceiver: TimeReceiver? = null
//定义一个时间广播的接收器
    inner class TimeReceiver : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent?) {
            if (intent != null) {
                desc = "$desc\n${DateUtil.nowTime} 收到一个${intent.action}广播"
                tv_system.text = desc
            }
        }
    }
}
```

分钟广播的监听结果如图 7-41 所示，当前时间正好是 13 时 25 分 0 秒，表示整分的时刻才会出现分钟广播。

complex

开始侦听分钟广播，请稍等。注意要保持屏幕亮着，才能正常收到广播
13:25:00 收到一个android.intent.action.TIME_TICK广播

图 7-41　接收到系统发送的分钟广播

7.5　实战项目：电商 App 的商品频道

常言道：人生若只如初见，说明对事物的第一印象是很重要的，这话对于电商 App 而言同样适用。卖东西凭的是什么？首先，不是凭卖家的自吹自擂，因为王婆卖瓜，自卖自夸，顾客可不是傻子。其次，也不是凭商品的标价，因为生意场上一分钱一分货，价格低了质量就没保障，价格高了顾客会货比三家。再次，名气也不一定可靠，因为名气大往往意味着故步自封、不思进取，像手机行业里的摩托罗拉、诺基亚等殷鉴不远。卖东西的关键条件之一是要抓住顾客的眼球，所谓百闻不如一见，你让顾客看着顺眼、觉得舒服，这单买卖就算不成，也还留下仁义。于是本节通过“电商 App 的商品频道”这个实战项目深入探讨如何在有限的屏幕空间内吸引用户的关注、激发用户的购买欲望。

7.5.1　需求描述

购物不分男女老少，每个群体的品位都大不一样，可是电商 App 的商品列表往往只有一个页面，要想在一个页面内展示不同的风格，可考虑根据频道分类来显示与该频道吻合的背景色。例如，图 7-42 所示的服装频道大多展示女士的裙装，此时的导航栏背景适合显示粉红背景；如图 7-43 所示的电器频道展示彩电、冰箱、洗衣机等家用电器，此时的导航栏背景更适合展示体现金属质感的蓝色背景。

不同商品的尺寸规则各不相同，像裙子分长裙和短裙，家电里面彩电比较宽而冰箱比较高，所以展示商品的时候，有的商品图片会宽一些，有的商品图片会高一些。对应这种长短不一的图片展示，就需要采取瀑布流效果的交错列表，具体如图 7-44 和图 7-45 所示，其中图 7-44 展示服装频道往上拉动后的界面，图 7-45 展示电器频道往上拉动后的界面。

另外，前面的商品页面允许往上拉动显示页面下方的商品，同时支持左右滑动来切换不同的商品频道，这样有了三个方向的手势，还剩一个往下拉动的手势可用来下拉刷新。用户在商品页面向下拉动时，如果已经拉到页面顶端还在下拉，则通常表示用户希望换一批商品更新页面。此时应当触发下拉刷新动作，在界面上提示商品页面正在刷新，提示效果如图 7-46 所示。接着电商 App 在商品列表顶端更换新的一批商品，此时完成下拉刷新的界面如图 7-47 所示。



图 7-42　电商 App 的服装频道界面



图 7-43　电商 App 的电器频道界面



图 7-44　服装频道上拉之后的界面



图 7-45　电器频道上拉之后的界面



图 7-46　服装频道正在刷新的界面



图 7-47　服装频道刷新完成的界面

如此一来，这个电商 App 的商品频道既考虑到各种顾客群体的视觉偏好，还充分兼顾有限屏幕空间与多样商品图片之间的相处，又同时支持上、下、左、右 4 个方向的滑动/滚动/拉动手势响应，花费许多心思，都是为了让顾客看着舒服、用着顺手。

7.5.2　开始热身：下拉刷新布局 SwipeRefreshLayout

电商 App 在商品列表页面往往提供下拉刷新功能，把列表页面整体下拉即可触发页面刷新操作。Android 为此提供了下拉刷新控件 SwipeRefreshLayout，可用于实现简单的下拉刷新功能。

下面是 SwipeRefreshLayout 的常用属性/方法说明。

- isRefreshing：该属性表示刷新的状态，true 表示正在刷新，false 表示结束刷新。注意，Kotlin 利用 isRefreshing 属性取代了原来的 setRefreshing 和 isRefreshing 两个方法。
- setOnRefreshListener：设置刷新监听器。需要重写监听器 OnRefreshListener 的 onRefresh 方法，该方法在下拉手势松开时触发。
- setColorSchemeColors：设置进度圆圈的圆环颜色列表。
- setProgressBackgroundColorSchemeColor：设置进度圆圈的背景颜色列表。
- setProgressViewOffset：设置进度圆圈的偏移量。第一个参数表示进度圈是否缩放，第二个参数表示进度圈开始出现时距顶端的偏移，第三个参数表示进度圈拉到最大时距顶端的偏移。

需要注意的是，SwipeRefreshLayout 节点下面只能有一个直接子视图，如果有多个直接子视图，那么只会展示第一个子视图，后面的子视图将不予展示。而且这个直接子视图还必须是允许滚动的控件，比如 ScrollView、ListView、GridView、RecyclerView、NestedScrollView 等，如果没有这些可滚动的视图，就无法支持下拉刷新操作。下面以仿微信公众号的消息列表为例给出 SwipeRefreshLayout 搭配 RecyclerView 的布局文件样例：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="5dp" >

    <android.support.v4.widget.SwipeRefreshLayout
        android:id="@+id/srl_dynamic"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <android.support.v7.widget.RecyclerView
            android:id="@+id/rv_dynamic"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="#aaaaff" />
    </android.support.v4.widget.SwipeRefreshLayout>
</LinearLayout>
```

上面仿公众号消息列表布局对应的 Kotlin 页面代码如下所示：

```
//由活动页面实现下拉刷新接口 OnRefreshListener
class SwipeRecyclerActivity : AppCompatActivity(), OnRefreshListener,
OnItemClickListener, OnItemLongClickListener, OnItemDeleteClickListener {
    lateinit var adapter: RecyclerSwipeAdapter
    private var currents = RecyclerInfo.defaultList
    private var alls = RecyclerInfo.defaultList

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        setContentView(R.layout.activity_swipe_recycler)
        //设置下拉刷新的监听器对象
        srl_dynamic.setOnRefreshListener(this)
        //设置刷新时转圈圈动画的渐变颜色列表
        srl_dynamic.setColorSchemeResources(R.color.red, R.color.orange,
R.color.green, R.color.blue)
        rv_dynamic.layoutManager = LinearLayoutManager(this)
        adapter = RecyclerSwipeAdapter(this, currents)
        adapter.setOnItemClickListener(this)
        adapter.setOnItemLongClickListener(this)
        adapter.setOnItemDeleteClickListener(this)
        rv_dynamic.adapter = adapter
        rv_dynamic.itemAnimator = DefaultItemAnimator()
        rv_dynamic.addItemDecoration(SpacesItemDecoration(1))
    }

    //监听器接口 OnRefreshListener 需要实现 onRefresh 方法完成刷新的事务处理
    override fun onRefresh() {
        mHandler.postDelayed(mRefresh, 2000)
    }

    private val mHandler = Handler()
    private val mRefresh = Runnable {
        //下拉刷新结束，要把 isRefreshing 设置为 false，以便从界面上去除转圈图标
        srl_dynamic.isRefreshing = false
        val position = (Math.random() * 100 % alls.size).toInt()
        val old_item = alls[position]
        val new_item = RecyclerInfo(old_item.pic_id, old_item.title,
old_item.desc)
        //每次刷新之时，往循环视图列表顶部添加一条信息
        currents.add(0, new_item)
        //通知循环适配器在第 0 项发生了添加操作
        adapter.notifyItemInserted(0)
        //让循环视图滚动到第 0 项的位置
        rv_dynamic.scrollToPosition(0)
    }

    //实现单项的点击方法
    override fun onItemClick(view: View, position: Int) {
        val desc = "您点击了第${position+1}项，标题是${currents[position].title}"
        toast(desc)
    }

    //实现单项的长按方法
    override fun onItemLongClick(view: View, position: Int) {
```

```
            //长按时在该项右边弹出删除按钮
            currents[position].pressed = !(currents[position].pressed)
            //通知循环适配器在第 position 项发生了变更操作
            adapter.notifyItemChanged(position)
        }

        //实现单项内部删除按钮的点击方法
        override fun onItemDeleteClick(view: View, position: Int) {
            //移除当前项
            currents.removeAt(position)
            //通知循环适配器在第 position 项发生了移除操作
            adapter.notifyItemRemoved(position)
        }
    }
```

上述仿公众号消息列表的下拉刷新效果如图 7-48～图 7-52 所示，其中图 7-48 所示为公众号列表的初始界面，图 7-49 所示为下拉列表触发刷新动作时的界面，图 7-50 所示为刷新完毕在列表顶部添加了新消息后的界面，图 7-51 所示为长按某消息弹出删除按钮时的界面，图 7-52 所示为点击删除按钮移除该项消息后的界面。



图 7-48　公众号列表的初始界面



图 7-49　下拉刷新时的等待界面



图 7-50　刷新结束添加新消息的列表



图 7-51　长按某项消息弹出删除按钮



图 7-52　点击删除按钮之后的消息列表

7.5.3　控件设计

商品频道的页面看起来既丰富又紧凑，其中运用了不少 Android 复杂控件，且待笔者细细列举如下。

（1）工具栏 Toolbar：页面顶部的导航栏，不用说就是工具栏 Toolbar。

（2）标签布局 TabLayout：频道页面顶部的“服装”和“电器”标签，用到了标签布局。

（3）翻页视图 ViewPager：与 TabLayout 配合使用，文本标签的切换对应着翻页视图的内部页面切换。

（4）碎片 Fragment：每一类的商品页面由相应的一个 Fragment 构成。

（5）循环视图 RecyclerView：在碎片布局内部，服装图片和电器图片的交错展示效果，运用了循环视图的瀑布流网格布局。

（6）循环适配器 RecyclerView.Adapter：每个商品的图片和名称经由循环适配器从而组装成完整的商品列表。

（7）碎片适配器 FragmentStatePagerAdapter：通过碎片适配器才能将几个碎片页封装进翻页视图。

（8）下拉刷新布局 SwipeRefreshLayout：循环视图借助下拉刷新布局方可触发下拉刷新操作。

除此之外，这个商品频道的实战项目还用到了广播 Broadcast，一旦用户选中具体的分类页面，则该分类页对应的碎片对象内部发出背景色变更的广播，然后注册了广播接收器的频道主页面接收到该广播，并把顶部导航栏的背景色改为新的颜色。

7.5.4　关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成商品频道项目，下面列举几个重要功能的 Kotlin 代码片段。

1. 关于频道页面的翻页适配器

翻页适配器的 Kotlin 编码本身挺简单，只需按照规矩重写 getItem、getCount、getPageTitle 三个方法就好了。这里所说的重点是，广播过滤器要求注册用的广播名称必须是常量，注意还必须是一开始就确定下来的编译时常量，故而适配器内部定义的广播名称 EVENT 字段务必要加上 const 修饰符，否则编译会提示失败。

下面是商品频道对应的 Kotlin 翻页适配器代码例子：

```
    class ChannelPagerAdapter(fm: FragmentManager, private val titles:
MutableList<String>) : FragmentPagerAdapter(fm) {

        //获取每个页面的碎片对象
        override fun getItem(position: Int): Fragment = when (position) {
            0 -> ClothesFragment()
            1 -> AppliancesFragment()
```

```
            else -> ClothesFragment()
        }

        //获取页面的数量
        override fun getCount(): Int = titles.size

        //获取页面的标题
        override fun getPageTitle(position: Int): CharSequence = titles[position]

        companion object {
            //广播过滤器的广播名称必须是编译时常量
            const val EVENT = "com.example.complex.adapter.ChannelPagerAdapter"
        }
    }
```

2. 关于循环视图的列表刷新

循环视图的列表数据发生变化之后，调用循环适配器的 notifyDataSetChanged 方法通知系统现在适配器列表发生了数据变更。然而这个处理还不够完善，假设现在循环视图的列表项已经占满整个屏幕，此时再往顶部添加一条新记录，就会感觉屏幕上的列表没有发生变化，也没看到插入动画。实际上循环视图顶部确实有添加新记录，把列表项往下拉就能看到，出现这个问题的原因是循环视图更新之后并不会自动下拉。要解决这个问题，得在 notifyDataSetChanged 方法调用之后，再调用循环视图对象的 scrollToPosition(0)方法，表示把列表项滚动到顶部的第一条记录。

下面是添加上述修改之后的 Kotlin 下拉刷新代码片段：

```
    override fun onRefresh() {
        //延迟两秒，模拟请求新一批商品的网络延时
        mHandler.postDelayed(mRefresh, 2000)
    }

    private val mHandler = Handler()
    private val mRefresh = Runnable {
        //下拉刷新结束，要把 isRefreshing 设置为 false，以便从界面上去除转圈图标
        srl_clothes.isRefreshing = false
        val i = alls.size - 1
        var count = 0
        while (count < 5) {
            val item = alls[i]
            alls.removeAt(i)
            alls.add(0, item)
            count++
        }
        //通知循环适配器发生了数据变更
        adapter.notifyDataSetChanged()
```

```
        //让循环视图滚动到第 0 项的位置
        rv_clothes.scrollToPosition(0)
    }
```

3. 关于碎片的选中事件

前面提到，翻页视图的选中事件可通过监听器 OnPageChangeListener 的 onPageSelected 方法来判断。但是，如果背景色变更的广播是从碎片内部发出来的，那么就得判断当前是哪一个碎片处于选中状态。本来按照常规，重写 Fragment 类的 setUserVisibleHint 方法即可，但这里有个问题：首次打开翻页视图时，默认显示第一个标签页，此时该标签页的生命周期为 onAttach->setUserVisibleHint->onCreateView，显然在这种情况之下，由于 setUserVisibleHint 方法在 onCreateView 方法之前调用，造成 App 还没来得及在 onCreateView 方法中给 ctx 变量赋值，因此这时候上下文对象为空，也就无法发送广播。为了避免上面说的意外情况，就要在 setUserVisibleHint 方法内部增加 ctx 变量是否为空的判断，只有上下文对象非空，才能继续向外发送广播。

下面便是 setUserVisibleHint 方法增加了非空判断的 Kotlin 碎片代码：

```
    override fun setUserVisibleHint(isVisibleToUser: Boolean) {
        super.setUserVisibleHint(isVisibleToUser)
        //如果该页是一打开的默认页，setUserVisibleHint 就先于 onCreateView 执行，此时 ctx
为空
        if (ctx != null) {
            val intent = Intent(ChannelPagerAdapter.EVENT)
            intent.putExtra("color", ctx!!.resources.getColor(R.color.pink))
            ctx!!.sendBroadcast(intent)
        }
    }
```

7.6 小　结

本章主要介绍了 Kotlin 如何实现几种复杂控件的调用，包括常见的几种视图排列（下拉框、列表视图、网格视图、循环视图）、新颖的材质设计（协调布局、工具栏、应用栏布局、可折叠工具栏布局）、页面切换的几种实现方式（翻页视图、碎片布局、标签布局）、Broadcast 广播组件的广播发送以及广播接收器的几种用法（临时广播、系统广播）。最后设计了一个实战项目“电商 App 的商品频道”，在该项目的 Kotlinn 编码中，采用了前面介绍的大部分布局和控件，以及 Broadcast 广播的发送和接收操作，另外还介绍了 Kotlin 对下拉刷新布局的用法。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会使用 Kotlin 操纵常见的视图排列，除了下拉框、列表视图、网格视图、循环视图的常规用法之外，重点掌握 Kotlin 对循环适配器的几项关键技术运用（lateinit 延迟初始化属性、LayoutContainer 布局容器插件以及函数参数在适配器中的应用）。

（2）学会使用 Kotlin 操纵新颖的材质设计，除了协调布局、工具栏、应用栏布局、可折叠工具栏布局的常规用法之外，还需了解并掌握支付宝首页头部伸缩的原理及其基础实现过程。

（3）学会使用 Kotlin 操纵页面切换的实现方式，除了翻页视图、碎片布局、标签布局的常规用法之外，重点复习引用相等的概念及其适用的场合。

（4）学会使用 Kotlin 进行 Broadcast 组件的广播发送和广播接收操作，其中重点了解两种常量的概念及其区别，并掌握修饰符 const 的使用场景。

（5）学会使用 Kotlin 操纵下拉刷新布局。

第 8 章

Kotlin 进行数据存储

本章介绍了 Android 四种主要存储方式的用法，包括共享参数 SharedPreferences、数据库 SQLite、文件 I/O 操作、App 的全局变量，另外介绍了安卓重要组件之一 Application 的常见用法。最后结合本章所学的知识演示了一个实战项目“电商 App 的购物车”的设计与实现。

8.1 使用共享参数 SharedPreferences

共享参数是安卓系统最简单的数据持久化存储方式，说它简单不只是因为存储结构简单，也是因为开发编码简单。即使通过 Java 编写共享参数读写的代码，其实不过寥寥几行，但要是鸡蛋里挑骨头，Java 代码当然不是那么完美。那么 Kotlin 究竟采取了哪些高科技手段，使得 Java 在 SharedPreferences 方面也得甘拜下风呢？接下来就好好探讨 Kotlin 对付共享参数的新技术、新手段。

8.1.1 共享参数读写模板 Preference

共享参数 SharedPreferences 是 Android 最简单的数据存储方式，常用于存取“Key-Value”键值对数据。在使用共享参数之前，首先要调用 getSharedPreferences 方法声明文件名与操作模式，对应的 Java 示例代码如下：

```
        SharedPreferences sps = getSharedPreferences("share",
Context.MODE_PRIVATE);
```

该方法的第一个参数是文件名，例子中的"share"表示当前的共享参数文件是 share.xml；第二个参数是操作模式，一般填 MODE_PRIVATE 表示私有模式。

共享参数若要存储数据，则需借助于 Editor 类，示例的 Java 代码如下：

```
        SharedPreferences.Editor editor = sps.edit();
        editor.putString("name", "阿四");
        editor.putInt("age", 25);
        editor.putBoolean("married", false);
        editor.putFloat("weight", 50f);
        editor.commit();
```

使用共享参数读取数据则相对简单，直接调用其对象的 get 方法即可获取数据，注意 get 方法的第二个参数表示默认值，示例的 Java 代码如下：

```
        String name = sps.getString("name", "");
        int age = sps.getInt("age", 0);
        boolean married = sps.getBoolean("married", false);
        float weight = sps.getFloat("weight", 0);
```

从上述数据读写的代码可以看出，共享参数的存取操作有些烦琐，因此实际开发中常将共享参数的相关操作提取到一个工具类，在新的工具类里面封装 SharedPreferences 的常用操作，下面便是一个共享参数工具类的 Java 代码例子：

```
public class SharedUtil {
    private static SharedUtil mUtil;
    private static SharedPreferences mShared;

    public static SharedUtil getIntance(Context ctx) {
        if (mUtil == null) {
            mUtil = new SharedUtil();
        }
        mShared = ctx.getSharedPreferences("share", Context.MODE_PRIVATE);
        return mUtil;
    }

    public void writeShared(String key, String value) {
        SharedPreferences.Editor editor = mShared.edit();
        editor.putString(key, value);
        editor.commit();
    }

    public String readShared(String key, String defaultValue) {
        return mShared.getString(key, defaultValue);
    }
}
```

有了共享参数工具类，外部读写 SharedPreferences 就比较方便了，比如下面的 Java 代码，无论是往共享参数写数据还是从共享参数读数据，均只要一行代码：

```
        //调用工具类写入共享参数
        SharedUtil.getIntance(this).writeShared("name", "阿四");
```

```
        //调用工具类读取共享参数
        String name = SharedUtil.getIntance(this).readShared("name", "");
```

当然这个工具类还有待进一步完善，因为它只支持字符串 String 类型的数据读写，并不支持整型、浮点数、布尔型等其他类型的数据读写。另外，如果外部需要先读取某个字段的数值，等处理完了再写回共享参数，那么使用该工具类也要两行代码（一行读数据、一行写数据），依旧有欠简洁。挑刺找毛病其实都是容易的，如果开发者仍然使用 Java 编码，那么能完善的就完善，不能完善的也不必苛求。

之所以挑 Java 实现方式的毛病，倒不是因为看它不顺眼整天吹毛求疵，而是因为 Kotlin 有更好的解决办法。为了趁热打铁方便比较两种方式的优劣，下面开门见山直接给出 Kotlin 封装共享参数的工具代码例子：

```
    class Preference<T>(val context: Context, val name: String, val default: T) :
ReadWriteProperty<Any?, T> {

        //通过属性代理初始化共享参数对象
        val prefs: SharedPreferences by lazy { context.getSharedPreferences
("default", Context.MODE_PRIVATE) }

        //接管属性值的获取行为
        override fun getValue(thisRef: Any?, property: KProperty<*>): T {
            return findPreference(name, default)
        }

        //接管属性值的修改行为
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
            putPreference(name, value)
        }

        //利用 with 函数定义临时的命名空间
        private fun <T> findPreference(name: String, default: T): T = with(prefs) {
            val res: Any = when (default) {
                is Long -> getLong(name, default)
                is String -> getString(name, default)
                is Int -> getInt(name, default)
                is Boolean -> getBoolean(name, default)
                is Float -> getFloat(name, default)
                else -> throw IllegalArgumentException("This type can be saved into
Preferences")
            }
            return res as T
        }

        private fun <T> putPreference(name: String, value: T) = with(prefs.edit()) {
            //putInt、putString 等方法返回 Editor 对象
```

```
            when (value) {
                is Long -> putLong(name, value)
                is String -> putString(name, value)
                is Int -> putInt(name, value)
                is Boolean -> putBoolean(name, value)
                is Float -> putFloat(name, value)
                else -> throw IllegalArgumentException("This type can be saved into
Preferences")
            }.apply() //commit 方法和 apply 方法都表示提交修改
        }
    }
```

外部在使用该工具类时，可在 Activity 代码中声明来自于 Preference 的委托属性，委托属性一旦声明，它的初始值便是从共享参数读取的数值，后续代码若给委托属性赋值，则立即触发写入动作，把该属性的最新值保存到共享参数中。于是外部操作共享参数的某个字段真正要书写的仅仅是下面的一行委托属性声明代码：

```
            //声明字符串类型的委托属性
            private var name: String by Preference(this, "name", "")
            //声明整型数类型的委托属性
            private var age: Int by Preference(this, "age", 0)
```

所谓百闻不如一见，赶紧运行 Kotlin 代码，看看它是否真的如传说中那般神奇。果不其然，Kotlin 通过 Preference 存取共享参数的结果与 Java 是一致的，具体的测试界面效果如图 8-1 和图 8-2 所示，其中图 8-1 展示数据保存到共享参数的结果，图 8-2 展示从共享参数读取数据的结果。



图 8-1　把注册信息保存到共享参数



图 8-2　从共享参数读取注册信息

8.1.2　属性代理等黑科技

既然 Kotlin 对共享参数的处理如此传神，那么读者肯定很好奇，这个高大上的 Preference 究竟运用了哪些黑科技呢？粗略数一下，大概用到了模板类、委托属性、lazy 修饰符、with 函数 4 种黑科技，且待笔者细细道来。

1. 模板类

因为共享参数允许保存的数据类型包括整型、浮点型、字符串等，所以要将 Preference 定义成模板类，具体的参数类型在调用时再指定。

除却代表模板类泛型的 T，该类中还有两个与之相似的元素，分别是 Any 和*，各自表示不同的含义。下面简单说明一下 T、Any 和*三者之间的区别。

（1）T 是抽象的泛型，在模板类中用来占位子，外部调用模板类时才能确定 T 的具体类型。

（2）Any 是 Kotlin 的基本类型，所有 Kotlin 类都从 Any 派生而来，故而它相当于 Java 里面的 Object。

（3）星号“*”表示一个不确定的类型，同样也是在外部调用时才能确定，这点跟 T 比较像。但 T 出现在模板类的定义中，而*与模板类无关，它出现在单个函数定义的参数列表中，因此星号相当于 Java 里面的问号“?”。

2. 委托属性/属性代理

注意到外部利用 Preference 声明参数字段时，后面跟着表达式“by Preference(...)”，这个 by 表示代理动作，在第 5 章的“5.3.5 接口代理”就介绍了如何让类通过关键字 by 实现指定接口的代理，当时举例说明给不同的鸟类赋予不同的动作。第 5 章的例子是接口代理（或称类代理），而这里则为属性代理，所谓属性代理，是说该属性的类型不变，但是属性的读写行为被后面的类接管了。

为什么需要接管属性的读写行为呢？举个例子，市民每个月都要交电费，自己每月跑去电力营业厅交钱显然够呛，于是后来支持在电力网站上自助缴费。然而上网缴费仍显麻烦，因为需要用户主动上网付费，要是用户忘记就不好办了。所以很多银行都推出了“委托代扣”的业务，只要用户跟银行签约并指定委托扣费的电力账户，那么在每个月指定时间，银行会自动从用户银行卡扣费并缴纳给指定的电力账户，如此省却了用户的人工操作。

现实生活中的委托扣费场景对应到共享参数这里，开发者的人工操作指的是：手工编码从 SharedPreferences 类读取数据和保存数据。而自动操作指的是给出一个约定：代理的属性自动通过模板类“Preference<T>”完成数据的读取和保存，也就是说，Preference<T>接管了这些属性的读写行为，接管后的操作即为模板类的 getValue 和 setValue 方法。因此，属性被接管的行为叫作属性代理，而被代理的属性称作委托属性。

3. lazy 修饰符

模板类 Preference<T>声明了一个共享参数的 prefs 对象，其中用到了关键字 lazy，lazy 的意思是懒惰，表示只在该属性第一次使用时执行初始化。联想到 Kotlin 还有类似的关键字名叫 lateinit，意思是延迟初始化，加上 lazy 可以归纳出 Kotlin 变量的三种初始化操作，具体说明如下。

（1）声明时赋值：这是最常见的变量初始化，在声明某个变量时，立即在后面通过等号“=”给它赋予具体的值。

（2）通过关键字 lateinit 延迟初始化：变量声明时没有马上赋值，但该变量仍是个非空变量，何时初始化由开发者编码决定。

（3）通过修饰符 lazy 在首次使用时初始化：声明变量时指定初始化动作，但该动作要等到变量第一次使用时才进行初始化。

此处的 prefs 对象使用 lazy 规定了属性值在首次使用时初始化，且初始化动作通过 by 后面的表达式来指定，即“{ context.getSharedPreferences("default", Context.MODE_PRIVATE) }”。连同大括号在内的这个表达式其实是个匿名实例，它内部定义了 prefs 对象的初始化语句，并返回 SharedPreferences 类型的变量值。

4. with 函数

with 函数的书写格式形如“with(函数头语句) { 函数体语句 }”，看这架势，with 方法的函数语句分为两部分，详述如下。

（1）函数头语句：头部语句位于紧跟 with 的圆括号内部。它先于函数体语句执行，并且头部语句返回一个对象，函数体语句在该对象的命名空间中运行。也就是说，体语句可以直接调用该对象的方法，而无须显式指定头部对象的实例名称。

（2）函数体语句：体语句位于常规的大括号内部。它要等头部语句处理完毕才会执行，同时体语句在头部语句返回对象的命名空间中运行。也就是说，体语句允许直接调用头部对象的方法，而无须显式指定该对象的实例名称。

综上所述，在模板类 Preference<T>的编码过程中，联合运用了 Kotlin 的多项黑科技，方才实现了优于 Java 的共享参数操作方式。

8.1.3 实现记住密码功能

第 6 章的实战项目“电商 App 的登录页面”，在页面下方有一个“记住密码”的复选框，当时只是为了演示控件 CheckBox 的运用，其实并未记住密码。用户退出后重新进入登录页面，App 并未自动填写上次的用户登录密码。现在利用前面介绍的模板类 Preference<T>对该项目进行改造，使之实现记住密码的功能。

改造过程若采用 Java 编码，则主要有以下三处改造内容：

（1）声明一个 SharedPreferences 对象，并在 onCreate 函数中调用 getSharedPreferences 方法对该对象进行初始化操作。

（2）登录成功时，如果用户勾选了“记住密码”，就使用共享参数保存手机号码与密码，即在 loginSuccess 函数中增加如下代码：

```
if (bRemember) {
    SharedPreferences.Editor editor = mShared.edit();
    editor.putString("phone", et_phone.getText().toString());
    editor.putString("password", et_password.getText().toString());
    editor.commit();
}
```

（3）在打开登录页面时，App 从共享参数中读取手机号码与密码，并展示在界面上，即在 onCreate 函数中增加如下代码：

```
String phone = mShared.getString("phone", "");
String password = mShared.getString("password", "");
```

```
        et_phone.setText(phone);
        et_password.setText(password);
```

同样的功能采取 Kotlin 编码，改造内容就很简单了，仅需在 Activity 代码中添加下面两行委托属性的声明语句：

```
        private var phone: String by Preference(this, "phone", "")
        private var password: String by Preference(this, "password", "")
```

由于上面的声明语句已经自动从共享参数获取属性值，接下来若要往共享参数保存新的属性值，只需修改委托属性的变量值即可。

修改完毕，不出意料的话，只要用户上次登录成功并且已勾选“记住密码”，那么下次进入登录页面时，App 就会自动填写上次登录的手机号码与密码。具体的效果图如图 8-3 和图 8-4 所示，其中图 8-3 所示为用户首次登录成功，此时已勾选“记住密码”复选框；图 8-4 所示为用户再次进入登录页面，因为上回登录成功时有记住密码，所以这次页面自动展示保存的登录信息。



图 8-3　首次登录成功时记住手机号和密码



图 8-4　再次登录时自动填写手机号和密码

8.2　使用数据库 SQLite

共享参数毕竟只能存储简单的键值对数据，如果需要存取更复杂的关系型数据，就要用到数据库 SQLite。虽然操作数据库的方法早已形成了一个固定的套路，但是 Java 的数据库编码依旧依赖于开发者的编程素养，无法完全做到面向业务进行数据库开发。而 Kotlin 并不墨守成规，努力屏蔽一些与业务无关的代码，使得利用 Kotlin 进行数据库编码更加安全可靠。下面一起来探个究竟，看看 Kotlin 引入了什么变化和改进。

8.2.1　数据库帮助器 SQLiteOpenHelper

尽管 SQLite 只是手机上的轻量级数据库，但它麻雀虽小、五脏俱全，与 Oracle 一样存在数据库的创建、变更、删除、连接等 DDL 操作，以及数据表的增、删、改、查等 DML 操作，因此开

发者对 SQLite 的使用编码一点都不能含糊。Android 为 SQLite 提供了两个管理类，分别是 SQLiteDatabase 和 SQLiteOpenHelper，二者的详细介绍如下。

1. SQLiteDatabase

SQLiteDatabase 是 SQLite 的数据库管理类，开发者可在 Activity 页面代码或者任何能取到 Context 的地方获取数据库实例，参考的 Java 代码如下所示：

```
        //创建数据库，如果已存在，就打开
        SQLiteDatabase db = getApplicationContext().openOrCreateDatabase
("test.db", Context.MODE_PRIVATE, null);
        //删除数据库
        getApplicationContext().deleteDatabase("test.db");
```

SQLiteDatabase 提供了若干操作数据表的 API，下面是它的常用方法说明。

- openDatabase：打开指定路径的数据库。
- isOpen：判断数据库是否已打开。
- close：关闭数据库。
- execSQL：执行拼接好的 SQL 控制语句。一般用于建表、删表、变更表结构。
- delete：删除符合条件的记录。
- update：更新符合条件的记录。
- insert：插入一条记录。
- query：执行查询操作，返回结果集的游标。
- rawQuery：执行拼接好的 SQL 查询语句，返回结果集的游标。

其中，可被 insert 和 update 方法直接使用的数据结构是 ContentValues 类，它类似于映射 Map，也提供了 put 和 get 方法用来存取键值对。区别在于，ContentValues 的键只能是字符串，查看 ContentValues 的源码会发现其内部保存键值对的数据结构就是 HashMap“private HashMap<String, Object> mValues;”。

另外，注意表的查询操作还要借助于游标类 Cursor 来实现，上面列举的方法中，query 和 rawQuery 两个查询方法返回的都是 Cursor 对象，那么获取查询结果就得根据游标的指示一条一条遍历结果集合。下面是游标 Cursor 类的常用方法。

（1）游标控制类方法，用于指定游标的状态。

- close：关闭游标。
- isClosed：判断游标是否关闭。
- isFirst：判断游标是否在开头。
- isLast：判断游标是否在末尾。

（2）游标移动类方法，把游标移动到指定位置。

- moveToFirst：移动游标到开头。
- moveToLast：移动游标到末尾。
- moveToNext：移动游标到下一个。

- moveToPrevious：移动游标到上一个。
- move：往后移动游标若干偏移量。
- moveToPosition：移动游标到指定位置。

（3）获取记录类方法，可获取记录的数量、类型以及取值。

- getCount：获取记录数。
- getInt：获取指定字段的整型值。
- getFloat：获取指定字段的浮点数值。
- getString：获取指定字段的字符串值。
- getType：获取指定字段的字段类型。

2. SQLiteOpenHelper

SQLiteDatabase 仅仅提供数据库的 DDL（数据定义）和 DML（数据管理）操作，并未提供完整的业务处理流程，这时 SQLiteOpenHelper 就派上用场了。SQLiteOpenHelper 是 SQLite 的使用帮助器，它是一个数据库操作的辅助工具，用于指导开发者合理使用 SQLite。要想在 App 开发中进行业务数据的保存和读取，必须按照以下步骤运用 SQLiteOpenHelper。

步骤 01 新建一个数据库操作类继承自 SQLiteOpenHelper，提示要重写 onCreate 和 onUpgrade 两个方法。其中，onCreate 方法只在第一次打开数据库时执行，在此可进行表结构创建的操作；而 onUpgrade 方法在数据库版本升高时执行，因此在 onUpgrade 函数内部，可以根据不同的新旧版本号进行表结构变更处理。

步骤 02 要封装保证数据库安全的必要方法，包括获取单例对象、打开数据库连接、关闭数据库连接等。

（1）获取单例对象：确保运行时数据库只被打开一次，避免重复打开数据库扔出异常。

（2）打开数据库连接：SQLite 也有锁机制，即读锁和写锁的处理，故而数据库连接也分为两种，读连接可调用 SQLiteOpenHelper 的 getReadableDatabase 方法获得，而写连接可调用 getWritableDatabase 获得。

（3）关闭数据库连接：数据库操作完毕，应当调用 SQLiteDatabase 对象的 close 方法关闭数据库连接。

步骤 03 提供对表记录进行增、删、改、查的操作方法。

由此可见，SQLiteOpenHelper 框架定义了一个数据库操作的代码准则，该准则需要开发者在编码的时候时刻注意遵守。一旦开发者遗忘某项操作，那么数据库处理极有可能会产生错误。

8.2.2　更安全的 ManagedSQLiteOpenHelper

8.2.1 小节提到，系统自带的 SQLiteOpenHelper 有个先天缺陷，就是它并未封装数据库管理类 SQLiteDatabase，这造成一个后果：开发者需要在操作表之前中手工打开数据库连接，然后在操作结束后手工关闭数据库连接。可是手工开关数据库连接存在着诸多问题，比如数据库连接是否重复打开了、数据库连接是否忘记关闭了、在 A 处打开数据库却在 B 处关闭数据是否造成业务异常。

以上的种种问题都制约了 SQLiteOpenHelper 的安全性。

鉴于此，Kotlin 结合 Anko 库推出了改良版的 SQLite 管理工具，名叫 ManagedSQLiteOpenHelper，该工具封装了数据库连接的开关操作，使得开发者完全无须关心 SQLiteDatabase 在何时、在何处调用，也就避免了手工开关数据库连接可能导致的各种异常。同时，ManagedSQLiteOpenHelper 的用法与 SQLiteOpenHelper 几乎一模一样，唯一的区别是：数据表的增、删、改、查语句需要放在 use 语句块之中，具体格式如下：

```
use {
    //1. 插入记录
    //insert(...)
    //2. 更新记录
    //update(...)
    //3. 删除记录
    //delete(...)
    //4. 查询记录
    //query(...)或者 rawQuery(...)
}
```

接下来，以用户注册信息数据库为例，看看 Kotlin 的数据库操作代码是怎样实现的，具体的实现代码示例如下：

```
    class UserDBHelper(var context: Context, private var DB_VERSION:
Int=CURRENT_VERSION) : ManagedSQLiteOpenHelper(context, DB_NAME, null, DB_VERSION) {
        companion object {
            private val TAG = "UserDBHelper"
            var DB_NAME = "user.db" //数据库名称
            var TABLE_NAME = "user_info" //表名称
            var CURRENT_VERSION = 1 //当前的最新版本，如有表结构变更，该版本号要加一
            private var instance: UserDBHelper? = null
            @Synchronized
            fun getInstance(ctx: Context, version: Int=0): UserDBHelper {
                if (instance == null) {
                    //如果调用时没传版本号，就使用默认的最新版本号
                    instance = if (version>0) UserDBHelper(ctx.applicationContext,
version)
                              else UserDBHelper(ctx.applicationContext)
                }
                return instance!!
            }
        }

        override fun onCreate(db: SQLiteDatabase) {
            Log.d(TAG, "onCreate")
            val drop_sql = "DROP TABLE IF EXISTS $TABLE_NAME;"
            Log.d(TAG, "drop_sql:" + drop_sql)
```

```
            db.execSQL(drop_sql)
            val create_sql = "CREATE TABLE IF NOT EXISTS $TABLE_NAME (" +
                "_id INTEGER PRIMARY KEY  AUTOINCREMENT NOT NULL," +
                "name VARCHAR NOT NULL," + "age INTEGER NOT NULL," +
                "height LONG NOT NULL," + "weight FLOAT NOT NULL," +
                "married INTEGER NOT NULL," + "update_time VARCHAR NOT NULL" +
                //演示数据库升级时要先把下面这行注释
                ",phone VARCHAR" + ",password VARCHAR" + ");"
            Log.d(TAG, "create_sql:" + create_sql)
            db.execSQL(create_sql)
        }

        override fun onUpgrade(db:SQLiteDatabase, oldVersion: Int, newVersion: Int) {
            Log.d(TAG, "onUpgrade oldVersion=$oldVersion, newVersion=
$newVersion")
            if (newVersion > 1) {
                //Android 的 ALTER 命令不支持一次添加多列，只能分多次添加
                var alter_sql = "ALTER TABLE $TABLE_NAME ADD COLUMN phone VARCHAR;"
                Log.d(TAG, "alter_sql:" + alter_sql)
                db.execSQL(alter_sql)
                alter_sql = "ALTER TABLE $TABLE_NAME ADD COLUMN password VARCHAR;"
                Log.d(TAG, "alter_sql:" + alter_sql)
                db.execSQL(alter_sql)
            }
        }

        //删除符合条件的记录
        fun delete(condition: String): Int {
            var count = 0
            use {
                count = delete(TABLE_NAME, condition, null)
            }
            return count
        }

        //添加一条记录
        fun insert(info: UserInfo): Long {
            val infoArray = mutableListOf(info)
            return insert(infoArray)
        }

        //删除多条记录
        fun insert(infoArray: MutableList<UserInfo>): Long {
            var result: Long = -1
            for (i in infoArray.indices) {
```

```
            val info = infoArray[i]
            var tempArray: List<UserInfo>
            // 如果存在同名记录，就更新记录
            // 注意条件语句的等号后面要用单引号括起来
            if (info.name.isNotEmpty()) {
                val condition = "name='${info.name}'"
                tempArray = query(condition)
                if (tempArray.size > 0) {
                    update(info, condition)
                    result = tempArray[0].rowid
                    continue
                }
            }
            // 如果存在同样的手机号码，就更新记录
            if (info.phone.isNotEmpty()) {
                val condition = "phone='${info.phone}'"
                tempArray = query(condition)
                if (tempArray.size > 0) {
                    update(info, condition)
                    result = tempArray[0].rowid
                    continue
                }
            }
            // 若不存在唯一性重复的记录，则插入新记录
            val cv = ContentValues()
            cv.put("name", info.name)
            cv.put("age", info.age)
            cv.put("height", info.height)
            cv.put("weight", info.weight)
            cv.put("married", info.married)
            cv.put("update_time", info.update_time)
            cv.put("phone", info.phone)
            cv.put("password", info.password)
            use {
                result = insert(TABLE_NAME, "", cv)
            }
            // 添加成功后返回行号，失败后返回-1
            if (result == -1L) {
                return result
            }
        }
        return result
    }

    //更新符合条件的记录
```

```
    @JvmOverloads
    fun update(info: UserInfo, condition: String = "rowid=${info.rowid}"): Int {
        val cv = ContentValues()
        cv.put("name", info.name)
        cv.put("age", info.age)
        cv.put("height", info.height)
        cv.put("weight", info.weight)
        cv.put("married", info.married)
        cv.put("update_time", info.update_time)
        cv.put("phone", info.phone)
        cv.put("password", info.password)
        var count = 0
        use {
            count = update(TABLE_NAME, cv, condition, null)
        }
        return count
    }

    //查询符合条件的记录
    fun query(condition: String): List<UserInfo> {
        val sql = "select rowid,_id,name,age,height,weight,married,update_time,
phone,password from $TABLE_NAME where $condition;"
        Log.d(TAG, "query sql: " + sql)
        var infoArray = mutableListOf<UserInfo>()
        use {
            val cursor = rawQuery(sql, null)
            if (cursor.moveToFirst()) {
                while (true) {
                    val info = UserInfo()
                    info.rowid = cursor.getLong(0)
                    info.xuhao = cursor.getInt(1)
                    info.name = cursor.getString(2)
                    info.age = cursor.getInt(3)
                    info.height = cursor.getLong(4)
                    info.weight = cursor.getFloat(5)
                    //SQLite 没有布尔型，用 0 表示 false，用 1 表示 true
                    info.married = if (cursor.getInt(6) == 0) false else true
                    info.update_time = cursor.getString(7)
                    info.phone = cursor.getString(8)
                    info.password = cursor.getString(9)
                    infoArray.add(info)
                    if (cursor.isLast) {
                        break
                    }
                    cursor.moveToNext()
```

```
                }
            }
            cursor.close()
        }
        return infoArray
    }

    //根据手机号码查询用户记录
    fun queryByPhone(phone: String): UserInfo {
        val infoArray = query("phone='$phone'")
        val info: UserInfo = if (infoArray.size>0) infoArray[0] else UserInfo()
        return info
    }

    //删除所有记录
    fun deleteAll(): Int = delete("1=1")

    //获取所有记录
    fun queryAll(): List<UserInfo> = query("1=1")
}
```

因为新来的管理类 ManagedSQLiteOpenHelper 来自于 Anko 库，所以要记得在 UserDBHelper 文件头部加入下面一行导入语句：

```
import org.jetbrains.anko.db.ManagedSQLiteOpenHelper
```

另外，有别于常见的 anko-common 包，Anko 库把跟数据库有关的部分放到了 anko-sqlite 包中，故而还需修改模块的 build.gradle 文件，在 dependencies 节点中补充下述的 anko-sqlite 包编译配置：

```
        compile "org.jetbrains.anko:anko-sqlite:$anko_version"
```

现在有了用户信息表的管理类，在 Activity 代码中存取用户信息就方便多了，下面是往数据库存储用户信息和从数据库读取用户信息的代码片段：

```
    var helper: UserDBHelper = UserDBHelper.getInstance(this)
    //往数据库存储用户信息
    btn_save.setOnClickListener {
        when (true) {
            et_name.text.isEmpty() -> toast("请先填写姓名")
            et_age.text.isEmpty() -> toast("请先填写年龄")
            et_height.text.isEmpty() -> toast("请先填写身高")
            et_weight.text.isEmpty() -> toast("请先填写体重")
            else -> {
                val info = UserInfo(name = et_name.text.toString(),
                age = et_age.text.toString().toInt(),
                height = et_height.text.toString().toLong(),
                weight = et_weight.text.toString().toFloat(),
```

```
                married = bMarried,
                update_time = DateUtil.nowDateTime)
                helper.insert(info)
                toast("数据已写入 SQLite 数据库")
            }
        }
    }

    //从数据库读取用户信息
    private fun readSQLite() {
        val userArray = helper.queryAll()
        var desc = "数据库查询到${userArray.size}条记录，详情如下："
        for (i in userArray.indices) {
            val item = userArray[i]
            desc = "$desc\n第${i+1}条记录信息如下：" +
                    "\n  姓名为${item.name}" +
                    "\n  年龄为${item.age}" +
                    "\n  身高为${item.height}" +
                    "\n  体重为${item.weight}" +
                    "\n  婚否为${item.married}" +
                    "\n  更新时间为${item.update_time}"
        }
        if (userArray.isEmpty()) {
            desc = "数据库查询到的记录为空"
        }
        tv_sqlite.text = desc
    }
```

以上代码对应的用户注册信息存取界面如图 8-5 和图 8-6 所示，其中图 8-5 所示为保存用户注册信息的效果图，图 8-6 所示为读取用户注册信息的效果图。

图 8-5　把注册信息保存到数据库



图 8-6　从数据库读取注册信息

8.2.3 优化记住密码功能

在前面的“8.1.3 实现记住密码功能”一节，利用共享参数实现了记住密码的功能。可是这个办法有个局限，就是它只能记住一个用户的登录信息，并且手机号码跟密码不存在匹配关系，如果换个手机号码登录，那么前一个用户的登录信息就被覆盖了。真正意义上的记住密码功能应该是先输入手机号码，然后根据手机号去匹配保存的密码，一个密码对应一个手机号码，如此方能实现具体号码的密码记忆功能。

现在通过运用数据库 SQLite 分条存储用户的登录信息，另外提供根据手机号码查找登录信息的方法，这样便可以同时记住多个手机号的密码。倘若使用 Java 编码，则具体的改造点主要有：

（1）声明一个 UserDBHelper 对象，然后在 onStart 方法中打开数据库连接，在 onStop 方法中关闭数据库连接，示例代码如下：

```
@Override
protected void onStart() {
    super.onStart();
    mHelper = UserDBHelper.getInstance(this, 2);
    mHelper.openWriteLink();
}

@Override
protected void onStop() {
    super.onStop();
    mHelper.closeLink();
}
```

（2）登录成功时，如果用户勾选了“记住密码”，就使用数据库保存手机号码与密码在内的登录信息，即在 loginSuccess 函数中增加以下 Java 代码：

```
    if (bRemember) {
        UserInfo info = new UserInfo();
        info.phone = et_phone.getText().toString();
        info.password = et_password.getText().toString();
        info.update_time = DateUtil.getNowDateTime("yyyy-MM-dd HH:mm:ss");
        mHelper.insert(info);
    }
```

（3）打开登录页面，用户输入手机号完毕，点击密码输入框时，App 从数据库中根据手机号查找登录记录，并将记录结果中的密码填入密码框。为了实现该步骤的功能，需要给密码框注册一个焦点变更监听器，就像下面一行代码那样：

```
    et_password.setOnFocusChangeListener(this);
```

这个焦点变更监听器要实现接口 OnFocusChangeListener，对应的事件处理方法是 onFocusChange，那么把数据库查询操作放在该方法里就行了，详细的 Java 代码如下所示：

```
        @Override
        public void onFocusChange(View v, boolean hasFocus) {
            String phone = et_phone.getText().toString();
            if (v.getId() == R.id.et_password) {
                if (phone.length() > 0 && hasFocus == true) {
                    UserInfo info = mHelper.queryByPhone(phone);
                    if (info != null) {
                        et_password.setText(info.password);
                    }
                }
            }
        }
```

从上面三个步骤看到，第一个步骤的打开和关闭数据库连接其实跟业务没什么关系，纯粹是装门面的劳民伤财之举。现在借用 Kotlin 的新工具 ManagedSQLiteOpenHelper 完全可以去掉数据库连接的步骤，于是 Kotlin 优化后的密码记住功能实际只要下面的步骤。

步骤 01 用户如果在登录页面勾选了“记住密码”，就需利用数据库保存包括手机号码与密码在内的登录信息，具体的 Kotlin 代码如下所示：

```
            if (bRemember) {
                val info = UserInfo(phone = et_phone.text.toString(),
                        password = et_password.text.toString(),
                        update_time = DateUtil.nowDateTime)
                mHelper.insert(info)
            }
```

步骤 02 如果用户上次登录成功时有勾选“记住密码”，那么下次打开 App 进入登录页面，一旦用户将手机号码输入完毕，App 就会到数据库找出与之匹配的密码，并自动填到密码框中。下面是实现自动匹配密码功能的 Kotlin 代码：

```
            et_password.setOnFocusChangeListener { v, hasFocus ->
                val phone: String = et_phone.text.toString()
                if (phone.isNotEmpty() && hasFocus) {
                    val info = mHelper.queryByPhone(phone)
                    et_password.setText(info.password)
                }
            }
```

步骤 03 代码改完了，再来看看登录页面的效果图，假定用户上次登录成功时选中“记住密码”，现在再次进入登录页面。如图 8-7 所示，此时用户输入手机号，光标还停留在手机框；接着用户点击密码框，光标随之跳到密码框，这时密码框自动填入了该手机号对应的密码串，如图 8-8 所示。



图 8-7　手机号码输入完毕的界面



图 8-8　根据手机号自动填写密码

8.3　文件 I/O 操作

尽管数据库能够存储大量的关系型数据，可它不是万能的，更多其他格式的数据仍然要以文件形式来保存。尽管 Java 的文件 I/O 功能已经足够强大，然而美中不足的是代码写起来很啰唆，仅仅一个写文件或者读文件操作都要通过字节流中转，编程新手往往很懵。现在 Kotlin 决心革除旧弊，把专业的文件 I/O 处理化繁为简，实现一行代码即可完成文件读写操作，赶快瞧一瞧 Kotlin 为新手们共享了哪些编程福利。

8.3.1　文件保存空间

手机上的存储空间分为内部存储和外部存储两部分，内部存储放的是手机系统以及各应用的安装目录，外部存储放的是公共文件，如图片、文档、音视频文件等。早期的外部存储被做成可插拔的 SD 卡，然而用户自己买的 SD 卡质量参差不齐，经常会影响 App 的正常运行，所以后来越来越多的手机把 SD 卡固化到手机内部，虽然拔不出来了，但是 Android 仍然称之为外部存储。由于内部存储空间有限，因此为了不影响系统的流畅运行，App 运行过程中要处理的文件都保存在外部存储空间。

为保证 App 正常读写外部存储，需在 AndroidManifest.xml 中增加 SD 卡的权限配置，具体的配置信息如下所示：

```
        <!-- SD 卡读写权限 -->
        <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
        <uses-permission android:name="android.permission.READ_EXTERNAL_STORAG"
/>
        <uses-permission
android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS" />
```

本来有了上面的权限配置，代码里面就能正常读写 SD 卡的文件。可是 Android 从 7.0 开始加强了 SD 卡的权限管理，即使 App 声明了完整的 SD 卡操作权限，系统仍然默认禁止该 App 访问外部存储。打开 7.0 系统的设置界面，进入具体应用的管理页面，会发现应用的存储功能被关闭了（指外部存储），如图 8-9 所示。

Storage
多媒体相关
读写手机存储
读写手机存储

图 8-9　Android 7.0 手机存储的权限设置界面

不过系统默认关闭存储其实只是关闭外部存储的公共空间，外部存储的私有空间依然可以正常读写。这是缘于 Android 把外部存储分成了两块区域，一块是所有应用均可访问的公共空间，另一块是只有应用自己才可访问的专享空间。之前说过，内部存储保存着每个应用的安装目录，但是安装目录的空间是很紧张的，所以 Android 在 SD 卡的“Android/data”目录下给每个应用又单独建了一个文件目录，用于给应用保存自己需要处理的临时文件。这个给每个应用单独建立的文件目录只有当前应用才能够读写文件，其他应用是不允许进行读写的，故而“Android/data”目录算是外部存储上的私有空间。这个私有空间本身已经做了访问权限控制，因此它不受系统禁止访问的影响，应用操作自己的文件目录就不成问题了。当然，因为私有的文件目录只有属主应用才能访问，所以一旦属主应用被用户卸载，那么对应的文件目录也会一起被清理掉。

既然外部存储分成了公共空间和私有空间两部分，这两部分空间的路径获取也就有所不同。获取公共空间的存储路径调用的是 Environment.getExternalStoragePublicDirectory 方法，获取应用私有空间的存储路径调用的是 getExternalFilesDir 方法。下面是分别获取两个空间路径的 Kotlin 代码例子：

```
    class FilePathActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_file_path)
            //获取系统的公共存储路径
            val publicPath = Environment.getExternalStoragePublicDirectory
(Environment.DIRECTORY_DOWNLOADS).toString();
            //获取当前 App 的私有存储路径
            val privatePath = getExternalFilesDir(Environment.
DIRECTORY_DOWNLOADS).toString();
            tv_file_path.text = "系统的公共存储路径位于${publicPath}" +
                    "\n\n 当前 App 的私有存储路径位于${privatePath}" +
                    "\n\nAndroid7.0 之后默认禁止访问公共存储目录"
        }
    }
```

该例子运行之后获得的路径信息如图 8-10 所示，可见应用的私有空间路径位于“外部存储根目录/Android/data/应用包名/files/Download”下。

storage

系统的公共存储路径位于/storage/emulated/0/Download

当前App的私有存储路径位于/storage/emulated/0/Android/data/com.example.storage/files/Download

Android7.0之后默认禁止访问公共存储目录

图 8-10 外部存储的两种访问路径

8.3.2 读写文本文件

Java 的文件处理用到了 io 库 java.io，该库虽然功能强大，但是与文件内容的交互还得通过输入输出流中转，致使文件读写操作颇为烦琐。因此，开发者通常得自己重新封装一个文件存取的工具类，以便在日常开发中调用。下面是一个文件工具类的简单 Java 代码：

```
public class FileUtil {

    //保存文本文件
    public static void saveText(String path, String txt) {
        try {
            FileOutputStream fos = new FileOutputStream(path);
            fos.write(txt.getBytes());
            fos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //读取文本文件
    public static String openText(String path) {
        String readStr = "";
        try {
            FileInputStream fis = new FileInputStream(path);
            byte[] b = new byte[fis.available()];
            fis.read(b);
            readStr = new String(b);
            fis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return readStr;
    }
}
```

从上述代码看到，仅仅是文本文件的内容保存和读取，就得规规矩矩写这么多行代码，并且还不太容易理解，对于新手来说着实不够友好。哪里有痛点，哪里就有优化，所以 Kotlin 在文件 API 这块也下了一番功夫，它以 Java 的 io 库为基础，利用扩展函数的功能添加了一些常用的文件内容读写方法，并且往往是一行代码便搞定功能，绝不拖泥带水。

比如把一段文本写入文本文件，只要调用 File 对象的 writeText 方法，即可实现写入文本的功能。真的只要一行代码，就像下面这样：

```
//把文本写入文件
File(file_path).writeText(content)
```

如此简洁又好用的代码，想必是许多开发者梦寐以求的。当然，Kotlin 同样支持其他格式的数据写入，前面的 writeText 方法是覆盖写入文本，若要往源文件追加文本，则可调用 appendText 方法。

看过了文件的写入操作，再来看看文件的读取操作。有了 writeText 方法带好头，Kotlin 又提供了以下几个好看且好用的文件内容读取方法。

- readText：读取文本形式的文件内容。
- readLines：按行读取文件内容。返回一个字符串的 List，文件有多少行，队列中就有多少个元素。

这几个方法理解起来毫不费力，从文件中读取全部的文本，也只要下面一行代码便成：

```
//读取文件的文本内容
val content = File(file_path).readText()
```

下面演示一下文本文件的读写效果，如图 8-11 所示，App 把页面录入的注册信息保存到 SD 卡上的文本文件；接着进入文件列表读取页面，选中某个文本文件，页面就展示该文件的文本内容，如图 8-12 所示。



图 8-11　把注册信息保存到文本文件



图 8-12　从文本文件读取注册信息

8.3.3　读写图片文件

像图片等二进制格式的文件，可通过字节数组的形式写入文件，Kotlin 提供了 writeBytes 方法用于覆盖写入字节数组，也提供了 appendBytes 方法用于追加字节数组。不过由于图像存储比较特殊，牵涉到压缩格式与压缩质量，因此还得通过输出流来处理（这是 Bitmap 的 compress 方法要求的），具体的图片文件写入代码如下所示：

```
fun saveImage(path: String, bitmap: Bitmap) {
    try {
        val file = File(path)
        //outputStream 获取文件的输出流对象
        //writer 获取文件的 Writer 对象
        //printWriter 获取文件的 PrintWriter 对象
        val fos: OutputStream = file.outputStream()
        //压缩格式为 JPEG 图像，压缩质量为 80%
        bitmap.compress(Bitmap.CompressFormat.JPEG, 80, fos)
        fos.flush()
        fos.close()
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

若想从图片文件中读取位图信息，按上面的 writeBytes 使用说明，应能调用 readBytes 方法。该办法确实可行，因为 Android 的位图工厂 BitmapFactory 刚好提供了 decodeByteArray 函数，用于从字节数组中解析位图，具体代码如下所示：

```
//方式一：利用字节数组读取位图
//readBytes 读取字节数组形式的文件内容
val bytes = File(file_path).readBytes()
//decodeByteArray 从字节数组解析图片
val bitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
```

之前提到将位图保存为图片文件时，通过输出流进行处理；那么反过来，从图片文件读取位图数据也可以通过输入流来完成。当然，多亏了 BitmapFactory 的 decodeStream 方法，使得输入流解析位图能够变成现实，以下便是输入流方式读取图片的代码例子：

```
//方式二：利用输入流读取位图
//inputStream 获取文件的输入流对象
val fis = File(file_path).inputStream()
//decodeStream 从输入流解析图片
val bitmap = BitmapFactory.decodeStream(fis)
fis.close()
```

前两种读取图片文件的方式其实都包含两个步骤：先从 File 对象获得文件内容，再利用位图工厂解码成位图。这么做也只需两行代码，但是不如读取文本的一行代码来得精炼，对于精益求精的开发者来说，此处仍然有着改善的空间。幸好位图工厂留了一手终极大招，名叫 decodeFile，只要给出图片文件的完整路径，文件读取和位图解析的操作都一齐搞定了，具体代码如下：

```
//方式三：直接从文件路径获取位图
//decodeFile 从指定路径解析图片
val bitmap = BitmapFactory.decodeFile(file_path)
```

真是想不到，只是从图片读取位图数据这个小功能就有至少三种方式，不但学到了 Kotlin 的

文件读取 API，而且温习了 Android 的 BitmapFactory 工具。开发者的口味各不相同，无论个人偏好哪种写法，以上三种方式总有一款适合你。

照例演示图片文件的读写结果，如图 8-13 所示，用户在注册页面录入注册信息，App 调用 getDrawingCache 方法把整个界面截图并保存到 SD 卡；然后在另一个页面的图片列表选择 SD 卡上的指定图片文件，页面就展示上次保存的截图图片，如图 8-14 所示。

图 8-13　把注册信息保存到图片文件

图 8-14　读取注册信息的图片文件

8.3.4　遍历文件目录

写文件和读文件是处理单个文件，没有太复杂的需求。倘若要求遍历某个目录下面的所有文本文件或者图片文件，那就麻烦了，因为该功能的需求点很丰富，例如要不要到子目录和孙子目录下搜索、文件跟文件夹都要匹配还是只匹配其中之一、筛选条件的文件扩展名都有哪些。想想这些详细的功能点都觉得头大，就算好不容易把符合条件的文件都挑出来，末了还得再来一个 for 循环进行处理操作。如果遍历功能采用 Java 编码，新手绝对无法自己写出实现代码，饶是高手也要颇费一番工夫。

现在有了 Kotlin 就方便多了，因为 Kotlin 把目录遍历这个功能重新梳理了一下，归纳为 FileTreeWalk 文件树，通过给文件树设置各式各样的参数与条件即可化繁为简，轻轻松松获取文件的搜索结果。文件树的使用很简单，首先调用 File 对象的 walk 方法得到 FileTreeWalk 实例，接着依次为该实例设置具体的条件，包括遍历深度、是否匹配文件夹、文件扩展名以及最后的文件队列循环处理。心动不如行动，快来看看 Kotlin 的文件遍历是怎么实现的，下面是搜寻指定目录下面所有文本文件的示例代码：

```
var fileNames: MutableList<String> = mutableListOf()
//在该目录下走一圈，得到文件目录树结构
val fileTree: FileTreeWalk = File(mPath).walk()
fileTree.maxDepth(1) //需遍历的目录层级为 1，即无须检查子目录
    .filter { it.isFile } //只挑选文件，不处理文件夹
```

```
            .filter { it.extension == "txt" } //选择扩展名为 txt 的文本文件
            .forEach { fileNames.add(it.name) } //循环处理符合条件的文件
```

注意到以上代码判断文件扩展名使用了“it.extension == "txt"”，如果符合条件的扩展名只有一个还好办，如果符合条件的扩展名有多个又该如何是好？譬如，图片文件的扩展名既可能是 png，也可能是 jpg，此时若用传统的“或”语句判断固然可行，但并不雅观。更好的办法是利用 Kotlin 的 in 条件，即判断文件的扩展名是否位于扩展名队列中，形如“it.extension in listOf("png","jpg")”这样，完整的图片文件搜索代码如下所示：

```
         var fileNames: MutableList<String> = mutableListOf()
        //在该目录下走一圈，得到文件目录树结构
        val fileTree: FileTreeWalk = File(mPath).walk()
        fileTree.maxDepth(1) //需遍历的目录层级为 1，即无须检查子目录
            .filter { it.isFile } //只挑选文件，不处理文件夹
            .filter { it.extension in listOf("png","jpg") } //选择扩展名为 png 和
jpg 的图片文件
            .forEach { fileNames.add(it.name) } //循环处理符合条件的文件
```

见识了 Kotlin 强大的文件操作 API，真让人耳目一新，如果你厌倦了 Java 的繁文缛节，不妨来 Kotlin 这里小试身手。

8.4 Application 全局变量

Application 是安卓的又一大组件，它的生命周期连接着 App 的整个运行过程，因此开发者常常给自定义的 Application 运用单例模式，使之具备全局变量的管理功能。那么 Kotlin 如何实现 Application 的单例化处理？又如何实现全局变量的定义和使用？本节接下来就对 Application 的这两个方面进行详细的介绍。

8.4.1 Application 单例化

在 App 运行过程中，有且仅有一个 Application 对象贯穿应用的整个生命周期，所以适合在 Application 中保存应用运行时的全局变量。而开展该工作的基础是必须获得 Application 对象的唯一实例，也就是将 Application 单例化。获取一个类的单例对象需要运用程序设计中常见的单例模式，通过 Java 编码实现单例化想必早已是大家耳熟能详的了。下面便是一个 Application 单例化的 Java 代码例子：

```
public class MainApplication extends Application {

    private static MainApplication mApp;

    public static MainApplication getInstance() {
        return mApp;
```

```
    }

    @Override
    public void onCreate() {
        super.onCreate();
        mApp = this;
    }
}
```

从以上代码可见，这个单例模式的实现过程主要有三个步骤，说明如下：

步骤 01 在自定义的 Application 类内部声明一个该类的静态实例。

步骤 02 重写 onCreate 方法，把自身对象赋值给第一步声明的实例。

步骤 03 提供一个供外部调用的静态方法 getInstance，该方法返回第一步声明的 Application 类实例。

无论是代码还是步骤，这个单例化的实现都还蛮简单的。同样的单例化过程通过 Kotlin 编码实现的话，静态属性和静态方法可利用伴生对象来实现，这样就形成了 Kotlin 单例化的第一种方式：手工声明属性的单例化，具体描述如下。

1. 手工声明属性的单例化

该方式与 Java 的常见做法一致，也是手工声明自身类的静态实例，然后通过静态方法返回自身实例。与 Java 的不同之处在于，Kotlin 引入了空安全机制，故而静态属性要声明为可空变量，然后获得实例时要在末尾加上双感叹号表示非空，当然也可事先将自身实例声明为延迟初始化属性。总之，两种声明手段都是为了确保一个目的，即 Application 类提供给外部访问的自身实例必须是非空的。

下面是手工单例化的 Kotlin 代码例子：

```
class MainApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        instance = this
    }

    //单例化的第一种方式：声明一个简单的 Application 属性
    companion object {
        //情况一：声明可空的属性
        private var instance: MainApplication? = null
        fun instance() = instance!!
        //情况二：声明延迟初始化属性
        //private lateinit var instance: MainApplication
        //fun instance() = instance
    }
}
```

2. 借助Delegates的委托属性单例化

第一种方式的单例化虽然提供了两种属性的声明手段，但只是为了保证自身实例的非空性。如果仅仅是确保属性非空，其实 Kotlin 已经提供了一个系统工具进行自动校验，这个工具便是Delegates的notNull方法。该方法返回非空校验的行为，只要将指定属性的读写行为委托给这个非空校验行为，开发者就无须手工进行非空判断。利用Delegates工具的属性代理功能就构成了Kotlin的第二种单例化方式。有关委托属性和属性代理的介绍可参考前面的“8.1.2 属性代理等黑科技”一节。

下面是利用系统代理行为实现单例化的Kotlin代码例子：

```
class MainApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        instance = this
    }

    //单例化的第二种方式：利用系统自带的Delegates生成委托属性
    companion object {
        private var instance: MainApplication by Delegates.notNull()
        fun instance() = instance
    }
}
```

第二种方式的委托属性单例化，在App代码中获取Application实例与第一种方式是一样的，都是调用“MainApplication.instance()”这个函数获得Application的自身实例。

3. 自定义代理行为的单例化

前两种单例化都只完成了非空校验，还不是严格意义上的单例化。真正的单例化是有且仅有一次赋值操作，尽管前两种单例化并未实现唯一赋值功能，不过在大多数场合已经够用了。可是作为孜孜不倦的开发者，务必要究根问底，到底能不能实现唯一赋值情况下的单例化？显然系统自带的Delegates工具没有提供大家期待的校验行为，于是开发者必须自己写一个能够校验赋值次数的行为类，目的是接管委托属性的读写行为。关于自定义接管行为的实现，本章一开头的“8.1.1 共享参数读写模板Preference”已给出了Preference<T>的完整源码，其中关键是重写读方法getValue和写方法setValue，因此在这里可借鉴Preference<T>完成自定义的委托行为编码。

下面是自定义代理行为的单例化代码：

```
class MainApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        instance = this
    }

    //单例化的第三种方式：自定义一个非空且只能一次性赋值的委托属性
    companion object {
        private var instance: MainApplication by NotNullSingleValueVar()
```

```
            fun instance() = instance
        }

        //定义一个属性管理类，进行非空和重复赋值的判断
        private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {
            private var value: T? = null
            override fun getValue(thisRef: Any?, property: KProperty<*>): T {
                return value ?: throw IllegalStateException("application not
initialized")
            }

            override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
                this.value = if (this.value == null) value
                else throw IllegalStateException("application already initialized")
            }
        }
    }
```

由上述代码看到，自定义的代理行为在 getValue 方法中进行非空校验，在 setValue 方法中进行重复赋值的校验，从而按照要求接管了委托属性的读写行为。

8.4.2　利用 Application 实现全局变量

8.4.1 小节介绍了如何实现 Application 对象的单例化，一旦有了单例的 Application 对象，就意味着 App 在运行过程中获取的 Application 实例是唯一的，因此可在该实例内部声明几个静态成员变量，从而形成所谓的全局变量。全局的意思就是其他代码都可以引用该变量，因此全局变量是共享数据和传递消息的好帮手。

适合在 Application 中保存的全局变量主要有下面几类数据：

（1）会频繁读取的信息，例如用户名、手机号等。

（2）从网络上获取的临时数据，为节约流量，也为减少用户等待时间，想暂时放在内存中供下次使用，例如应用 logo、商品图片等。

（3）容易因频繁分配内存而导致内存泄漏的对象，例如处理器 Handler、线程池 ThreadPool 等。

要想通过 Application 实现全局变量的读写，得完成以下几个步骤：

步骤 01　写一个类 MainApplication 继承自 Application，该类要采用单例模式，内部声明自身的一个静态单例对象，在创建 App 时把自身赋值给这个静态实例，然后提供一个访问该静态对象的 instance 函数。

步骤 02　在 Activity 中调用 MainApplication 的 instance 方法，获得 MainApplication 的一个静态对象，便可通过该对象访问 MainApplication 的公共变量和公共方法。

步骤 03　不要忘了在 AndroidManifest.xml 中注册新定义的 Application 类名，即在 application 节点中增加 android:name 属性，其值为".MainApplication"，注册信息举例如下：

```
    <application
        android:name=".MainApplication"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name" >
```

下面演示 Application 的全局变量读写效果，如图 8-15 所示，App 把注册信息保存到 MainApplication 的全局变量中；然后在另一个页面，又从 MainApplication 的全局变量中读取之前保存的注册信息，如图 8-16 所示。

storage
姓名：老刘
年龄：50
身高：165
体重：60
婚否：已婚
保存到全局内存

图 8-15　把注册信息保存到全局变量



图 8-16　从全局变量读取注册信息

8.5　实战项目：电商 App 的购物车

购物车的应用面很广，凡是电商 App 都可以看到它的身影，这里选中购物车作为本章的实战项目，除了它使用广泛的特点外，更是因为它用到了多种存储方式。现在就让我们开启电商购物车的体验之旅吧。

8.5.1　需求描述

先来看看常见的购物车长什么模样，第一次进入购物车频道，购物车里面是空的，如图 8-17 所示。接着去商场频道选购手机，随便挑选几部手机加入购物车，然后返回购物车页面，即可看到车里的商品列表，如图 8-18 所示，有商品图片、名称、数量、单价、总价等信息。当然，购物车并不只是展示代购商品，还要支持最终购买的结算操作、删除不想要的商品、清空购物车等功能。

购物车的存在感是很强的，并不仅仅在购物车页面才能看到购物车。往往在商场频道，甚至某个商品详情页面，都会看到某个角落冒出个购物车图标。一旦有新商品加入购物车，那么购物车图标上的商品数量立马加一。当然，用户也可以点击购物车图标直接跳转到购物车页面。如图 8-19 所示，商场频道除了商品列表之外，页面右上角还有一个购物车图标，这个图标有时在页面右上角，有时在页面右下角，总之会有一个地方存放购物车图标。商品详情页面通常也有购物车图标，如图 8-20 所示，倘若用户在详情页面把商品加入购物车，那么图标上的数字也会加一。

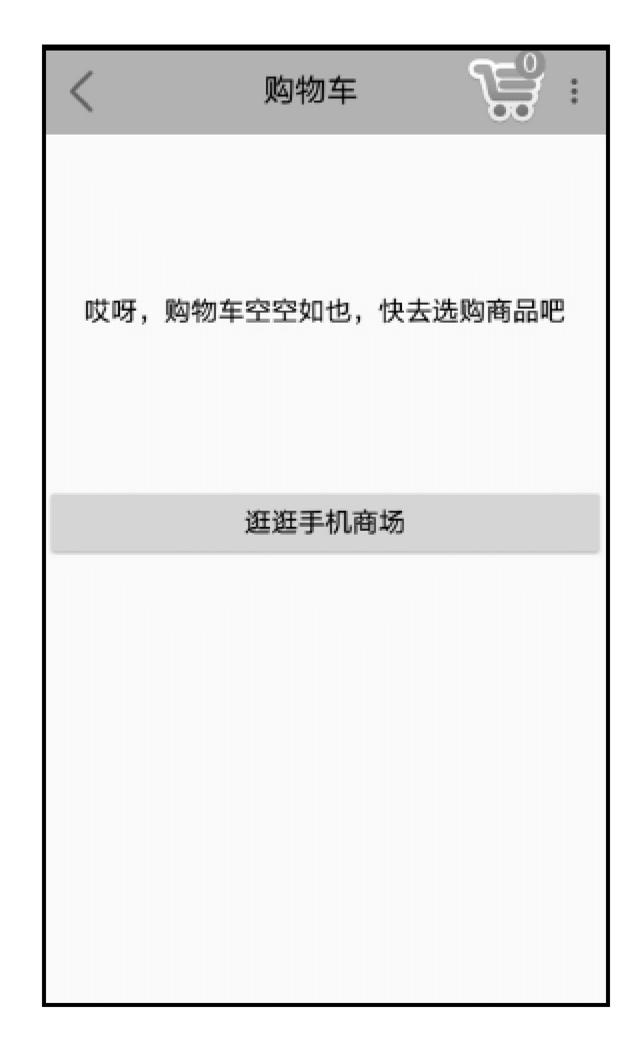


图 8-17　首次打开购物车页面



图 8-18　选购商品后的购物车

至此，大概过了一遍购物车需要实现的基本功能，提需求总是很简单的，真正落到实处还得开发者发挥想象力，把购物车做成一个功能完备的模块。



图 8-19　手机商场频道页面



图 8-20　手机商品详情页面

8.5.2　开始热身：选项菜单 OptionsMenu

之前的章节在进行某项控制操作时，一般是由按钮控件触发。如果页面上需要支持多个控制操作，比如去商场购物、清空购物车、查看商品详情、删除指定商品等，就得往页面上添加多个按钮。如此一来，App 页面显得杂乱无章，满屏的按钮既碍眼又不便操作。这时，就要请选项菜单 OptionsMenu 来帮忙了。选项菜单平时不显示在界面上，只有在用户按下手机右下角的菜单键，或者按下工具栏右上角的三点键时，才会弹出一个菜单列表供用户选择操作。

若要实现选项菜单的功能，首先要定义一个菜单的布局文件，就像每个 Activity 页面都有一个

布局文件一样。不同的是，页面的布局文件放在 res/layout 目录下，而菜单的布局文件放在 res/menu 目录下。下面是一个菜单布局文件 menu_option.xml 的例子，很简单，就是 menu 与 item 的组合排列：

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_change_time"
        android:orderInCategory="1"
        android:title="改变时间"/>

    <item
        android:id="@+id/menu_change_color"
        android:orderInCategory="8"
        android:title="改变颜色"/>

    <item
        android:id="@+id/menu_change_bg"
        android:orderInCategory="9"
        android:title="改变背景"/>
</menu>
```

有了上面的菜单描述布局，接着在 Activity 代码中重写 onCreateOptionsMenu 方法，指定当前页面加载该菜单布局。加载菜单布局的 Kotlin 代码如下所示：

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.menu_option, menu)
    return true
}
```

最后，重写 Activity 代码里面的 onOptionsItemSelected 方法，对各个菜单项进行对应的分支处理。下面是一个对选项菜单列表做多路分支的 Kotlin 代码例子：

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.menu_change_time -> setRandomTime()
        R.id.menu_change_color -> tv_option.setTextColor(randomColor)
        R.id.menu_change_bg -> tv_option.setBackgroundColor(randomColor)
    }
    return true
}

private fun setRandomTime() {
    val desc = "${DateUtil.nowDateTime} 这里是菜单显示文本"
    tv_option.text = desc
}

private val mColorArray = intArrayOf(Color.BLACK, Color.WHITE, Color.RED,
```

```
Color.YELLOW, Color.GREEN, Color.BLUE, Color.CYAN, Color.MAGENTA, Color.GRAY,
Color.DKGRAY)
        private val randomColor: Int
            get() {
                val random = (Math.random() * 10 % 10).toInt()
                return mColorArray[random]
            }
```

代码书写完毕，即可按下右下角的菜单键，也可按下右上角的三点键，这两种方式均会弹出选项菜单。二者不同的是，菜单键在屏幕下方弹出菜单列表，如图 8-21 所示；而三点键在屏幕右上方弹出菜单列表，如图 8-22 所示。



图 8-21　按菜单键弹出菜单列表

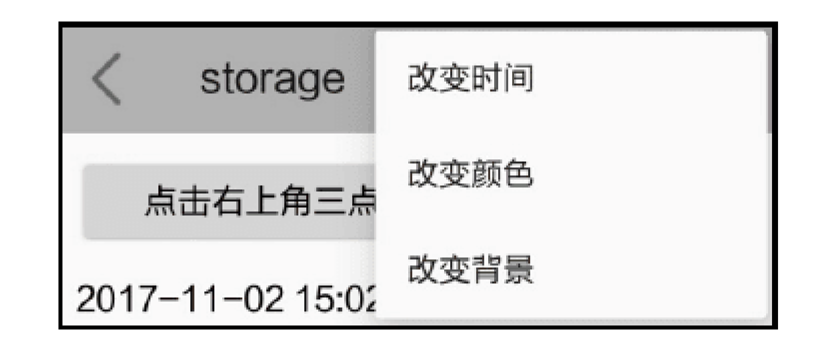


图 8-22　按三点键弹出菜单列表

8.5.3　控件设计

首先来找找看，购物车到底采取了哪些存储方式。

- 数据库 SQLite：最直观的肯定是数据库了，购物车里的商品列表一定是放在 SQLite 中的，增删改查都少不了它。
- 共享参数 SharedPreferences：注意到不同页面右上角的购物车图标都有数字，表示购物车中的商品数量，这个商品数量建议保存在共享参数中。因为每个页面都要显示商品数量，如果每次都到数据库中执行 count 操作，就会很消耗资源。并且商品数量需要持久化存储，所以不适合放在全局内存中，不然下次启动 App，内存中的变量又是从 0 开始。
- SD 卡文件：通常情况下，商品图片来自于电商平台的服务器，这年头流量是很宝贵的，可是图片恰恰很耗流量（尤其是大图），于是从用户的钱包着想，App 得把好不容易下载来的图片保存在 SD 卡。这样一来，下次用户再访问商品详情页面时，App 便能直接从 SD 卡获取商品图片，不但不花费流量而且加快浏览速度，一举两得。
- 全局内存：访问 SD 卡的图片文件固然是个好主意，然而像商场频道、购物车频道都有可能在一个页面上展示多张商品小图，如果每张小图都要访问 SD 卡，频繁的 SD 读写操作也蛮耗资源的。更好的办法是把商品小图加载进全局内存，这样直接从内存中获取图片，高效又快速。之所以不把商品大图也放入全局内存，是因为大图很耗空间，一不小心便会占用几十兆内存。

然后考虑一下几个页面的排版布局，主要用到 Android 的以下几个控件。

- 工具栏 Toolbar：购物车、商场频道、商品详情这几个页面需要统一风格，故界面顶部采取 Toolbar 作为整体的导航栏。

- 列表视图 ListView：购物车中的商品列表从上到下依次排列，适合使用 ListView 来展示商品列表。
- 网格视图 GridView：商场频道页面的商品陈列橱柜，通过 GridView 能够最大限度地利用屏幕空间。
- 循环视图 RecyclerView：无论是 ListView 还是 GridView，都存在着记录项点击与内部控件点击的冲突问题，因此最好采用 RecyclerView 代替 ListVIew 和 GridView。
- 选项菜单 OptionsMenu：购物车页面除了展示商品列表外，还要支持前往商场频道、清空购物车、返回上个页面等功能。要是这些功能都使用按钮操作，势必弄得页面拥挤不堪，所以要做成菜单列表形式，在用户需要的时候再在界面上弹出菜单，这个菜单列表的弹出效果如图 8-23 所示。
- 提醒对话框 AlertDialog：若要删除购物车中的指定商品，可通过监听长按事件来触发。当然，为了避免用户误操作，对于长按事件需要弹出提示框，好让用户确认是否真的取消购买该商品，提示框的显示效果如图 8-24 所示。只有用户确定不再购买商品，方可从购物车列表删除该商品。



图 8-23　购物车页面的菜单列表



图 8-24　取消购买商品的提示框

8.5.4　关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成购物车项目，下面列举几个重要功能的 Kotlin 代码片段。

1. 关于页面跳转

因为购物车页面允许直接跳到商场频道页面，并且商场频道页面也允许跳到购物车页面，所以如果用户在这两个页面之间跳来跳去，然后按返回键，就会发现返回的时候也会在这两个页面间往返跳转。造成该问题的原因是：每次启动活动页面都往活动栈加入一个新活动，那么返回出栈时，也只好一个一个活动依次退出了。

解决该问题的办法参见第 6 章的“6.4.3 跳转时指定启动模式”，对于活动跳转需要指定标志 FLAG_ACTIVITY_CLEAR_TOP，表示活动栈有且仅有该页面的唯一实例，如此即可避免多次返回同一页面的情况。若是利用 Kotlin 设置这个启动标志，则可调用 clearTop 方法予以实现，下面是两个页面之间跳转的 Kotlin 代码例子：

```
//在购物车页面跳到商场频道页面，通过 clearTop 函数设置启动标志
btn_shopping_channel.setOnClickListener {
```

```
        startActivity(intentFor<ShoppingChannelActivity>().clearTop())
    }
    //在商场频道页面跳到购物车页面，通过 clearTop 函数设置启动标志
    iv_cart.setOnClickListener {
        startActivity(intentFor<ShoppingCartActivity>().clearTop())
    }
```

2. 关于菜单列表

考虑到用户使用习惯，建议把选项菜单功能集成到工具栏上面，也就是在工具栏右侧展示系统自带的竖排三点键。一旦用户点击三点键，就在屏幕右上角弹出菜单列表供用户选择，此时选项菜单又称为溢出菜单 OverflowMenu，意思是导航的工具栏不够放了，溢出来了。

下面是溢出菜单的菜单布局例子，暂时包含三个菜单项：

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_shopping"
        android:orderInCategory="1"
        android:title="去商场购物"/>

    <item
        android:id="@+id/menu_clear"
        android:orderInCategory="2"
        android:title="清空购物车"/>

    <item
        android:id="@+id/menu_return"
        android:orderInCategory="9"
        android:title="返回"/>
</menu>
```

与该菜单布局对应的 Kotlin 响应菜单项的点击代码如下所示：

```
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_cart, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.menu_shopping -> startActivity(intentFor<
ShoppingChannelActivity>().clearTop())
            R.id.menu_clear -> {
                mCartHelper.deleteAll() //清空购物车数据库
                showCount(0)
                toast("购物车已清空")
```

```
            }
            R.id.menu_return -> finish()
        }
        return true
    }
```

3. 关于商品图片的缓存

通常，商品图片由后端服务器提供，App 打开页面时再从服务器下载所需的商品图。可是购物车模块的多个页面都会展示商品图片，如果每次都到服务器加载图片，显然既耗时间又耗流量，非常不经济。因此，App 开发会把常用的图片进行缓存处理，图片一旦从服务器下载成功，便在手机存储空间上保存图片文件。然后下次界面需要加载商品图片时，就先从手机寻找该图片，如果找到，就读取图片的位图信息，否则再到服务器下载图片。

以上的缓存逻辑是最简单的二级缓存，实际开发往往使用更高级的三级缓存机制，即“运行内存→外部存储（SD 卡）→网络下载”。其中，运行内存的速度快但容量小，所以更适合存放频繁访问的商品小图；而 SD 卡速度稍慢但容量大，所以适合存放不太经常访问的商品大图。按此思路构建购物车的商品图片缓存框架，对应的 Kotlin 代码示例如下：

```
    companion object {
        //模拟网络数据，初始化数据库中的商品信息
        fun downloadGoods(ctx: Context, isFirst: String, helper: GoodsDBHelper) {
            val path = MainApplication.instance().getExternalFilesDir
(Environment.DIRECTORY_DOWNLOADS).toString() + "/"
            Log.d(TAG, "path=$path")
            if (isFirst == "true") {
                val goodsList = GoodsInfo.defaultList
                for (i in goodsList.indices) {
                    val info = goodsList[i]
                    val rowid = helper.insert(info)
                    info.rowid = rowid
                    //往运行内存写入商品小图
                    val thumb = BitmapFactory.decodeResource(ctx.resources,
info.thumb)
                    MainApplication.instance().mIconMap.put(rowid, thumb)
                    val thumb_path = "$path${rowid}_s.jpg"
                    FileUtil.saveImage(thumb_path, thumb)
                    info.thumb_path = thumb_path
                    //往 SD 卡保存商品大图
                    val pic = BitmapFactory.decodeResource(ctx.resources,
info.pic)
                    val pic_path = "$path${rowid}.jpg"
                    FileUtil.saveImage(pic_path, pic)
                    pic.recycle()
                    info.pic_path = pic_path
                    helper.update(info)
```

```
                }
            } else {
                val goodsArray = helper.queryAll()
                for (item in goodsArray) {
                    Log.d(TAG, "rowid=${item.rowid}, thumb_path=
${item.thumb_path}")
                    val thumb = BitmapFactory.decodeFile(item.thumb_path)
                    MainApplication.instance().mIconMap.put(item.rowid, thumb)
                }
            }
        }
    }
```

4. 关于购物车的商品列表

虽然商品列表看起来很适合运用列表视图 ListView，但实际上采取循环视图 RecyclerView 更为合适。其中的缘由是 RecyclerView 功能更强大、画面更柔和，而且循环适配器的 Kotlin 编码更简单。下面是一个购物车列表的 Kotlin 适配器代码例子：

```
    class RecyclerCartAdapter(context: Context, private val carts:
MutableList<CartInfo>) : RecyclerBaseAdapter<RecyclerView.ViewHolder>(context) {

        override fun getItemCount(): Int = carts.size

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
            val view: View = inflater.inflate(R.layout.item_recycler_cart, parent,
false)
            return ItemHolder(view)
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val vh: ItemHolder = holder as ItemHolder
            vh.bind(carts[position], itemClickListener, itemLongClickListener)
        }

        class ItemHolder(override val containerView: View?) : RecyclerView.
ViewHolder(containerView), LayoutContainer {
            fun bind(item: CartInfo,
                     clickListener: RecyclerExtras.OnItemClickListener?,
                     longClickListener: RecyclerExtras.OnItemLongClickListener?) {
                iv_thumb.setImageBitmap(MainApplication.instance().
mIconMap[item.goods_id])
                tv_name.text = item.goods.name
                tv_desc.text = item.goods.desc
                tv_count.text = item.count.toString()
```

```
                tv_price.text =  item.goods.price.toString()
                tv_sum.text = (item.count * item.goods.price).toString()
                // 列表项的点击事件需要自己实现
                ll_item.setOnClickListener { v ->
                    clickListener?.onItemClick(v, position)
                }
                ll_item.setOnLongClickListener { v ->
                    longClickListener?.onItemLongClick(v, position)
                    true
                }
            }
        }
    }
```

5. 关于长按事件的提示框

这个提示框采取 Anko 库提供的 alert 扩展函数，可有效缩小代码数量，同时增强代码可读性。使用 alert 方法的 Kotlin 代码例子如下所示：

```
        override fun onItemLongClick(view: View, position: Int) {
            val cart = mCartArray[position]
            val message = "尊敬的用户，您是否不再购买${cart.goods.name}? "
            alert(message, "商品购买提示") {
                positiveButton("不再购买") {
                    //从购物车删除商品的数据库操作
                    mCartHelper.delete("goods_id=" + cart.goods_id)
                    //更新购物车中的商品数量
                    showCount(mCount - cart.count)
                    toast("已从购物车删除${cart.goods.name}")
                    //刷新购物车的商品列表
                    showCart()
                }
                negativeButton("我再想想") { }
            }.show()
        }
```

8.6 小　　结

本章主要介绍了 Kotlin 操作 Android 的几种数据存储方式，包括利用 Preference<T>实现共享参数的键值对信息存取、利用 ManagedSQLiteOpenHelper 实现更安全的数据库记录管理、通过全新的文件 I/O 函数库简化文件处理、Application 组件的单例化及全局变量的实现。最后设计了一个实战项目“电商 App 的购物车”，通过该项目的编码，进一步复习巩固了本章 4 种存储方式的使用，另外介绍了选项菜单的基本用法。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会利用工具类 Preference<T>进行共享参数的键值对管理工作，并掌握委托属性、lazy 修饰符、with 函数的基本用法。

（2）学会使用 Kotlin 的 ManagedSQLiteOpenHelper 工具进行数据库操作编码。

（3）学会通过 Kotlin 的文件 I/O 函数库进行文件相关处理，包括文本文件读写、图片文件读写、文件目录遍历等。

（4）学会按照 Kotlin 的编码风格实现 Application 的单例化，并通过单例 Application 操作全局变量。

（5）学会 Kotlin 编码实现选项菜单的调用。

第 9 章

Kotlin 自定义控件

本章介绍了 Android 三种常见的自定义控件形式，包括自定义普通视图、自定义简单动画、自定义通知栏，另外介绍了安卓四大组件之一的服务 Service 的常见用法。最后结合本章所学的知识演示一个实战项目“电商 App 的生鲜团购”的设计与实现。

9.1　自定义普通视图

Android 提供了丰富多彩的视图与控件，已经能够满足大部分的业务需求，然而计划赶不上变化，总是有意料之外的情况需要特殊处理。比如第 7 章在“7.3.1 翻页视图 ViewPager”小节提到了翻页标题栏 PagerTabStrip，该控件无法在布局文件中指定文本大小和文本颜色，只能在代码中调用 setTextSize 和 setTextColor 方法进行设置。这用起来殊为不便，如果它能像 TextView 那样直接在布局中指定文本大小和颜色就好了，要想让控件 PagerTabStrip 支持该功能，就得通过自定义视图来实现。本节将对自定义视图的三个步骤（构造对象、测量尺寸、绘制部件）进行详细介绍，并说明如何使用 Kotlin 完成新视图的自定义处理。

9.1.1　构造对象

自定义视图的第一种途径是自定义属性，仍旧以翻页标题栏 PagerTabStrip 举例，现在给它新增两个自定义属性，分别是文本颜色 textColor 和文本大小 textSize。接下来给出自定义属性对应的 Java 编码步骤。

步骤 01 在 res\values 目录下创建属性定义文件 attrs.xml，文件内容如下所示，其中 declare-styleable 的 name 属性值表示新视图的名称，两个 attr 节点表示新增的两个属性分别是 textColor 和 textSize：

```
<resources>
    <declare-styleable name="CustomPagerTab">
        <attr name="textColor" format="color" />
        <attr name="textSize" format="dimension" />
    </declare-styleable>
</resources>
```

步骤 02 在模块源码的 com.example.custom.widget 包下创建 CustomPagerTab.java，填入以下自定义视图的 Java 代码：

```
public class CustomPagerTab extends PagerTabStrip {
    private int textColor = Color.BLACK;
    private int textSize = 15;

    public CustomPagerTab(Context context) {
        super(context);
    }

    public CustomPagerTab(Context context, AttributeSet attrs) {
        super(context, attrs);
        //构造函数从 attrs.xml 读取 CustomPagerTab 的自定义属性
        if (attrs != null) {
            TypedArray attrArray=getContext().obtainStyledAttributes(attrs,
R.styleable.CustomPagerTab);
            //从布局文件中获取新属性 textColor 的数值
            textColor =
attrArray.getColor(R.styleable.CustomPagerTab_textColor, textColor);
            //从布局文件中获取新属性 textSize 的数值
            textSize =
attrArray.getDimensionPixelSize(R.styleable.CustomPagerTab_textSize, textSize);
            attrArray.recycle();
        }
        //应用布局文件的 textColor 文本颜色
        setTextColor(textColor);
        //应用布局文件的 textSize 文本大小
        setTextSize(TypedValue.COMPLEX_UNIT_SP, textSize);
    }

//    //PagerTabStrip 没有三个参数的构造函数
//    public PagerTab(Context context, AttributeSet attrs, int defStyleAttr) {
//    }
}
```

步骤 03 在布局文件的根节点增加自定义的命名空间声明，如“xmlns:app="http://schemas.android.com/apk/res-auto"”，并把 android.support.v4.view.PagerTabStrip 的节点名称改为自定义视图的全路径名称，如“com.example.custom.widget.PagerTab”，同时在该节点下指定新增的两个属性，即 app:textColoe 与 app:textSize。修改之后的布局文件示例如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="10dp" >

    <android.support.v4.view.ViewPager
        android:id="@+id/vp_content"
        android:layout_width="match_parent"
        android:layout_height="400dp" >

        <com.example.custom.widget.CustomPagerTab
            android:id="@+id/pts_tab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:textColor="@color/red"
            app:textSize="17sp" />
    </android.support.v4.view.ViewPager>
</LinearLayout>
```

完成以上三步修改后，运行测试应用，展示的界面效果如图 9-1 所示，此时翻页标题栏的文字颜色变为红色，而且字体也变大了。



图 9-1　自定义翻页标题栏的文字效果

上述自定义属性的操作一共有三个步骤，其中第二个步骤涉及 Java 代码，接下来利用 Kotlin 改写 CustomPagerTab 类的代码，主要改动有以下两点：

（1）原来的两个构造函数合并为带默认参数的一个主构造函数，并且主构造函数直接跟在类名后面定义。

（2）类名后面要补充注解“@JvmOverloads constructor”，表示该类支持被 Java 代码调用。因为 xml 布局文件中声明了自定义视图的节点，而系统是通过 SDK 里的 Java 代码找到自定义视图类的，所以凡是自定义视图都要加上该注解，否则 App 运行时会抛出异常。

下面是 CustomPagerTab 类改写之后的 Kotlin 定义代码：

```
//自定义视图要在类名后面增加“@JvmOverloads constructor”，因为布局文件中的自定义视图必须兼容 Java
class CustomPagerTab @JvmOverloads constructor(context: Context, attrs: AttributeSet?=null) : PagerTabStrip(context, attrs) {
    private var txtColor = Color.BLACK
    private var textSize = 15
    init {
        //初始化时从 attrs.xml 读取 CustomPagerTab 的自定义属性
```

```
            if (attrs != null) {
                val attrArray = getContext().obtainStyledAttributes(attrs,
R.styleable.CustomPagerTab)
                //从布局文件中获取新属性 textColor 的数值
                txtColor = attrArray.getColor(R.styleable.
CustomPagerTab_textColor, txtColor)
                //从布局文件中获取新属性 textSize 的数值
                textSize = attrArray.getDimensionPixelSize
(R.styleable.CustomPagerTab_textSize, textSize)
                attrArray.recycle()
            }
            //应用布局文件的 textColor 文本颜色
            setTextColor(txtColor)
            //应用布局文件的 textSize 文本大小
            setTextSize(TypedValue.COMPLEX_UNIT_SP, textSize.toFloat())
        }
    }
```

9.1.2 测量尺寸

从 9.1.1 小节看到，自定义视图的主构造函数主要有两个用途，一个是读取布局文件中的自定义属性值，另一个是进行初始化设置。但定义构造函数仅仅是自定义视图的一部分，完整的自定义视图编码由以下三部分组成：

（1）定义构造函数，读取自定义属性值并进行初始化设置。

（2）重写测量函数 onMesure，计算该视图的宽高尺寸。

（3）重写绘图函数 onDraw 或者 dispatchDraw，在当前视图内部绘制指定形状。

以上自定义视图编码的三个组成部分，第一部分的构造函数已经在 9.1.1 小节介绍过了，接下来的两个小节继续介绍测量函数与绘图函数的重载实现。

一个视图的宽和高其实在页面布局的时候就决定了，视图节点的 android:layout_width 属性指定了该视图的宽度，而 android:layout_height 属性指定了该视图的高度。这两个属性又有三种取值方式，分别是：取值 match_parent 表示与上级视图一样尺寸，取值 wrap_content 表示按照自身内容的实际尺寸，最后一种则直接指定了具体的 dp 数值。在多数情况下，系统按照这三种取值方式完全能够自动计算正确的视图宽度和视图高度。

当然也有例外，像列表视图 ListView 就是个另类，尽管 ListView 在多数场合的高度计算也不会出错，但是把它放到 ScrollView 中便出现问题了。ScrollView 本身叫作滚动视图，而列表视图 ListView 也是可以滚动的，于是一个滚动视图嵌套另一个也能滚动的视图，那么在双方的重叠区域，上下滑动的手势究竟表示要滚动哪个视图？这个滚动冲突的问题不只令开发者脑袋糊涂，便是 Android 系统也得神经错乱。所以 Android 目前的处理对策是：如果 ListView 的高度被设置为 wrap_content，此时列表视图就只显示一行的高度，然后布局内部只支持滚动 ScrollView。

如此，虽然滚动冲突的问题暂时解决，但是又带来一个新问题，好好的列表视图仅仅显示一

行内容，这让出不了头的剩余列表行情何以堪？按照用户正常的思维逻辑，列表视图应该显示所有行，并且列表内容要跟着整个页面一起向上或者向下滚动。显然，此时系统对 ListView 的默认处理方式并不符合用户习惯，只能对其进行改造使之满足用户的使用习惯。改造列表视图的一个可行方案是重写它的测量函数 onMesure，无论布局文件中设定的视图高度是多少，都把列表视图的高度改为最大高度，即所有列表项高度加起来的总高度。

根据以上思路自定义一个扩展自 ListView 的不滚动列表视图 NoScrollListView，它的 Java 实现代码如下所示：

```
public class NoScrollListView extends ListView {

    public NoScrollListView(Context context) {
        super(context);
    }

    public NoScrollListView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public NoScrollListView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    @Override
    public void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        //将高度设为最大值，即所有项加起来的总高度
        int expandSpec = MeasureSpec.makeMeasureSpec(Integer.MAX_VALUE >> 2,
MeasureSpec.AT_MOST);
        super.onMeasure(widthMeasureSpec, expandSpec);
    }
}
```

看到上面的 Java 代码一口气写了三个构造函数，明显很啰嗦，要是改写成以下的 Kotlin 代码，只要一个主构造函数就够了：

```
//自定义视图要在类名后面增加“@JvmOverloads constructor”，因为布局文件中的自定义视图
必须兼容 Java
class NoScrollListView @JvmOverloads constructor(context: Context, attrs:
AttributeSet?=null, defStyle: Int=0) : ListView(context, attrs, defStyle) {

    //将高度设为最大值，即所有项加起来的总高度
    public override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec:
Int) {
        //注意位运算符的写法，按位右移在 Kotlin 中使用运算符 shr 来表达
        val expandSpec = MeasureSpec.makeMeasureSpec(Integer.MAX_VALUE shr 2,
MeasureSpec.AT_MOST)
        super.onMeasure(widthMeasureSpec, expandSpec)
    }
}
```

接下来，为了方便演示改造前后列表视图的界面效果对比，在一个页面布局中放入 ScrollView 节点，然后在该节点下面同时添加 ListView 节点和自定义的 NoScrollListView 节点，布局文件的内容示例如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:padding="50dp"
                android:text="下面是系统自带的 ListView"
                android:textColor="@color/red"
                android:textSize="17sp" />

            <ListView
                android:id="@+id/lv_planet"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginBottom="50dp"
                android:dividerHeight="1dp" />

            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:padding="50dp"
                android:text="下面是全部展开的 ListView"
                android:textColor="@color/green"
                android:textSize="17sp" />

            <com.example.custom.widget.NoScrollListView
                android:id="@+id/nslv_planet"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginBottom="50dp"
```

```
            android:dividerHeight="1dp" />
        </LinearLayout>
    </ScrollView>
</LinearLayout>
```

回到演示页面的 Activity 代码，给 ListView 和 NoScrollListView 两个控件对象设置一模一样的行星列表数据，具体实现的 Kotlin 代码如下所示：

```
class MeasureViewActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_measure_view)
        //lv_planet 是系统自带的 ListView，被 ScrollView 嵌套只能显示一行
        val adapter1 = PlanetAdapter(this, Planet.defaultList)
        lv_planet.adapter = adapter1
        lv_planet.onItemClickListener = adapter1
        lv_planet.onItemLongClickListener = adapter1
        //nslv_planet 是自定义控件 NoScrollListView，会显示所有行
        val adapter2 = PlanetAdapter(this, Planet.defaultList)
        nslv_planet.adapter = adapter2
        nslv_planet.onItemClickListener = adapter2
        nslv_planet.onItemLongClickListener = adapter2
    }
}
```

重新编译运行 App，然后上下滑动测试页面，即可观察到两种列表的区别。如图 9-2 所示，这是测试页面的初始界面，此时系统自带的 ListView 仅显示一行内容，而开发者自定义的 NoScrollListView 显示多行内容。接着把测试页面往上拉动，滚动后的界面如图 9-3 所示，此时系统自带的 ListView 带着仅有的一行完全向上滚没了，而开发者自定义的 NoScrollListView 随着上拉手势持续滚动，可见 NoScrollListView 内部的列表项完完全全地展示了出来。



图 9-2 系统自带的 ListView 嵌套效果

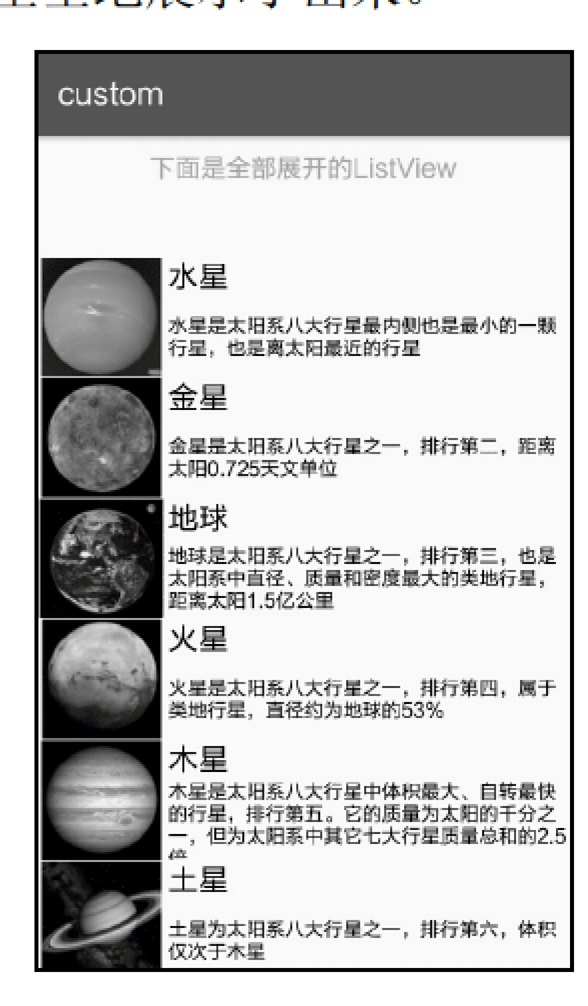


图 9-3 自定义的的 ListView 展开效果

9.1.3　绘制部件

自定义视图的第三个步骤是重写绘图函数，绘图函数包括 onDraw 和 dispatchDraw 两种，二者的区别是：onDraw 既出现在控件类视图又出现在布局类视图，而 dispatchDraw 只出现在布局类视图。假设一个布局文件包含一个线性布局 LinearLayout 节点，且 LinearLayout 节点下又包含一个文本视图 TextView 节点，则它们之间的绘图函数调用顺序依次为：线性布局的 onDraw→文本视图的 onDraw→线性布局的 dispatchDraw。这个绘图次序意味着线性布局在 onDraw 函数中绘制的画面很可能被后来的文本视图涂鸦所覆盖，但最终定稿的却是线性布局在 dispatchDraw 函数中的绘图结果。借用"螳螂捕蝉，黄雀在后"的成语类比，此时线性布局的 onDraw 函数是蝉，文本视图的 onDraw 函数是螳螂，线性布局的 dispatchDraw 函数是黄雀。

讲完了 onDraw 与 dispatchDraw 两个函数之间的次序关系，也就弄清楚了两种绘图函数分别适用的场合，即控件视图只能重写 onDraw 函数，而布局视图若不想绘图效果被下级控件覆盖，则必须重写 dispatchDraw 函数。下面举一个自定义文本视图的例子，在当前文本视图的四周绘制圆角矩形，对应的 Java 实现代码如下所示：

```
public class RoundTextView extends TextView {

    public RoundTextView(Context context) {
        super(context);
    }

    public RoundTextView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public RoundTextView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    //控件只能重写 onDraw 方法
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        //通过画笔 Paint 在画布 Canvas 上绘制图案
        Paint paint = new Paint();
        paint.setColor(Color.RED); //设置画笔的颜色
        paint.setStrokeWidth(2); //设置线条的宽度
        paint.setStyle(Style.STROKE); //设置画笔的风格，STROKE 表示空心
        paint.setAntiAlias(true); //设置画笔为无锯齿
        RectF rectF = new RectF(1, 1, this.getWidth()-1, this.getHeight()-1);
        //方法 drawRoundRect 表示绘制圆角矩形
        canvas.drawRoundRect(rectF, 10, 10, paint);
    }
}
```

接着使用 Kotlin 改写自定义的圆角文本视图 RoundTextView，则合并了三个构造函数的 Kotlin 代码如下所示：

```
    //自定义视图要在类名后面增加“@JvmOverloads constructor”
    class RoundTextView @JvmOverloads constructor(context: Context, attrs:
AttributeSet?=null, defStyle: Int=0) : TextView(context, attrs, defStyle) {

        //控件只能重写 onDraw 方法
        override fun onDraw(canvas: Canvas) {
            super.onDraw(canvas)
            //通过画笔 Paint 在画布 Canvas 上绘制图案
            val paint = Paint()
            paint.color = Color.RED //设置画笔的颜色，即红色
            paint.strokeWidth = 2f //设置线条的宽度
            paint.style = Style.STROKE //设置画笔的风格，STROKE 表示空心
            paint.isAntiAlias = true //设置画笔为无锯齿
            val rectF = RectF(1f, 1f, (this.width - 1).toFloat(), (this.height - 1).
toFloat())
            //方法 drawRoundRect 表示绘制圆角矩形
            canvas.drawRoundRect(rectF, 10f, 10f, paint)
        }
    }
```

下面再举一个自定义线性布局的例子，同样在当前线性布局的四周绘上圆角矩形，对应的 Java 实现代码如下：

```
    public class RoundLayout extends LinearLayout {

        public RoundLayout(Context context) {
            super(context);
        }

        public RoundLayout(Context context, AttributeSet attrs) {
            super(context, attrs);
        }

        public RoundLayout(Context context, AttributeSet attrs, int defStyle) {
            super(context, attrs, defStyle);
        }

        //布局一般重写 dispatchDraw 方法，防止绘图效果被上面的控件覆盖
        @Override
        protected void dispatchDraw(Canvas canvas) {
            super.dispatchDraw(canvas);
            Paint paint = new Paint();
            paint.setColor(Color.BLUE);
            paint.setStrokeWidth(2);
            paint.setStyle(Style.STROKE);
```

```
            paint.setAntiAlias(true);
            RectF rectF = new RectF(1, 1, this.getWidth()-1, this.getHeight()-1);
            canvas.drawRoundRect(rectF, 10, 10, paint);
        }
    }
```

依然使用 Kotlin 改写自定义的圆角布局 RoundLayout，改写后的 Kotlin 代码示例如下：

```
    //自定义视图要在类名后面增加“@JvmOverloads constructor”
    class RoundLayout @JvmOverloads constructor(context: Context, attrs:
AttributeSet?=null, defStyle: Int=0) : LinearLayout(context, attrs, defStyle) {

        //布局一般重写 dispatchDraw 方法，防止绘图效果被上面的控件覆盖
        override fun dispatchDraw(canvas: Canvas) {
            super.dispatchDraw(canvas)
            val paint = Paint()
            paint.color = Color.BLUE //设置画笔的颜色，即蓝色
            paint.strokeWidth = 2f //设置线条的宽度
            paint.style = Style.STROKE //设置画笔的风格，STROKE 表示空心
            paint.isAntiAlias = true //设置画笔为无锯齿
            val rectF = RectF(1f, 1f, (this.width - 1).toFloat(), (this.height -
1).toFloat())
            //方法 drawRoundRect 表示绘制圆角矩形
            canvas.drawRoundRect(rectF, 10f, 10f, paint)
        }
    }
```

在接下来的演示案例中，联合运用新定义的圆角文本视图 RoundTextView 与圆角布局 RoundLayout，通过外层的 RoundLayout 嵌套内层的 RoundTextView，以便观察内外层的两个圆角矩形。下面是演示用到的布局文件内容：

```
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="10dp" >

        <com.example.custom.widget.RoundLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="10dp"
            android:orientation="vertical" >

            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:padding="10dp"
```

```
                android:gravity="center"
                android:text="入群须知"
                android:textColor="@color/black"
                android:textSize="25sp" />

            <com.example.custom.widget.RoundTextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:padding="10dp"
                android:text="新入群的朋友请注意，入群时先向大家问好，然后自报姓名、性别、
年龄、身高、体重、职业，最重要的一点是：自己上传照片！"
                android:textColor="@color/black"
                android:textSize="17sp" />
        </com.example.custom.widget.RoundLayout>
    </LinearLayout>
```

嵌套圆角矩形的演示界面如图 9-4 所示，可见所有文本的外层被一个蓝色的圆角矩形所环绕，表示这是圆角布局 RoundLayout 所绘制的；而新入群注意事项的文本周围则是一个红色的圆角矩形，表示这是圆角文本视图 RoundTextView 所绘制的。

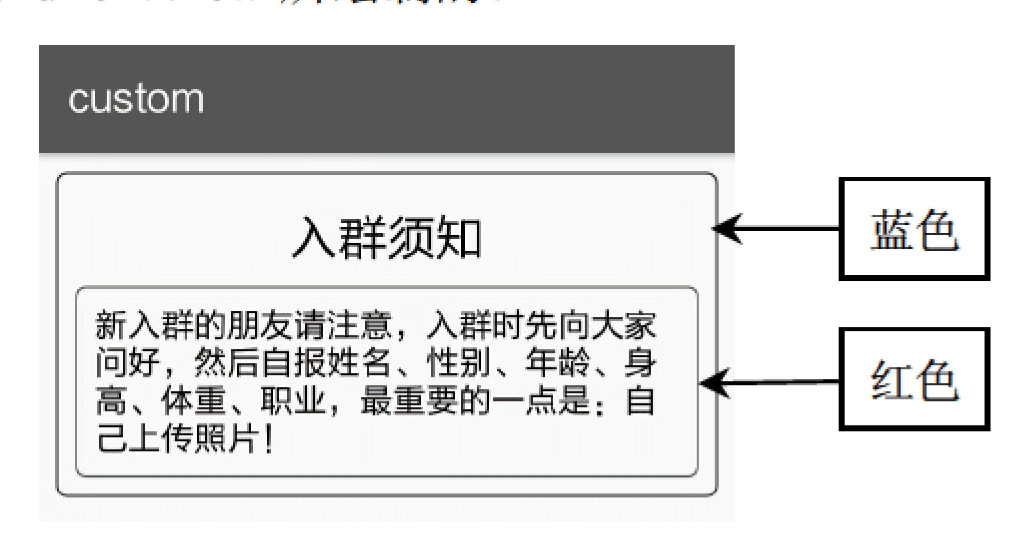


图 9-4　圆角布局和圆角文本视图的显示效果

9.2　自定义简单动画

手机 App 是给人民大众使用的，所以不只是要求功能方面能够正常运行，也要求界面上的用户体验足够美观活泼。想想看，一个静止不动的应用界面明显缺乏变化，流于僵硬；相比之下，一个动态展示的应用界面既让人觉得流畅又让人觉得舒服。动态展示往往用到动画技术，对于简单的动画效果来说，只需要短时间的间隔，然后持续刷新界面即可。这里的持续刷新动作便是本节将要讲述的任务 Runnable，除此之外，本节还将介绍如何利用任务 Runnable 结合进度条 ProgressBar 实现一个简单的进度条动画。

9.2.1　任务 Runnable

任务 Runnable 定义了一个可以独立运行的代码片段，通常用于界面控件的延迟处理，比如有

时为了避免同时占用某种资源造成冲突，有时则是为了反复间隔刷新界面从而产生动画效果。运行一个任务也有多种形式，既能在 UI 线程中调用处理器对象的 post 或者 postDelayed 方法，也能另外开启分线程来执行 Runnable 对象。在运行任务之前，必须事先声明该任务的对象，然后才能由调用者执行该任务。Kotlin 代码声明 Runnable 对象有 4 种方式，分别对应不同的业务场景，接下来就依次阐述 Runnable 对象在 Kotlin 编码中的 4 种声明方式：内部类、匿名内部类、简化类实例、匿名实例。

1. 内部类

内部类方式是最循规蹈矩的，在代码里先书写一个继承自 Runnable 的内部类，再重写它的 run 方法，填入具体的业务逻辑处理。以最常见的计数器为例，每隔一秒便在界面上显示加一后的计数结果，使用内部类方式进行演示的话，就是以下的 Kotlin 代码例子：

```
private val handler = Handler()
private var count = 0
//inner 修饰符表示这是一个内部类
inner private class Counter : Runnable {
    override fun run() {
        count++
        tv_result.text = "当前计数值为：$count"
        handler.postDelayed(this, 1000)
    }
}
```

然后在 Activity 页面的按钮点击事件中加入下面一行 Kotlin 代码，在点击按钮时触发这个计数任务：

```
handler.post(Counter())
```

运行测试应用，界面上的计数效果如图 9-5 和图 9-6 所示，其中图 9-5 表示当前正在计数，图 9-6 表示当前停止计数，终止的计数值为 12。



图 9-5 计数任务开始计数



图 9-6 计数任务停止计数

2. 匿名内部类

内部类的方式最正规，无疑也是最啰唆的。由于这个计数任务仅仅在点击按钮时启动一次，因此并不需要对其显式构造，只要在定义内部类时顺便声明该任务的实例即可。此时的任务定义代码便从内部类方式变成了匿名内部类方式，采取 Kotlin 编码的话，注意使用关键字 object 占位，表示这是一个匿名内部类，完整的 Kotlin 任务定义代码如下所示：

```
//使用关键字 object 占位，表示这是一个匿名内部类
private val counter = object : Runnable {
    override fun run() {
        count++
        tv_result.text = "当前计数值为：$count"
        handler.postDelayed(this, 1000)
    }
}
```

因为定义内部类的同时就声明了任务实例，所以处理器直接运行该实例即可启动计数 任务：

```
    handler.post(counter)
```

内部类与匿名内部类这两种方式其实内部都拥有类的完整形态，故而它们的 run 方法允许使用关键字 this 指代自身的任务对象，于是示例代码中的“handler.postDelayed(this, 1000)”表示间隔一秒之后重复执行自身任务。正因为能够重复执行任务，所以这两种方式可用于持续刷新界面的动画效果。

3. 简化类实例

前面两种内部类实现方式拥有类的完整形态，意味着必须显式重写 run 方法，可是这个任务类肯定只能重写 run 方法，即使开发者不写出来，run 方法也是逃不掉的。在第 1 章，当时为了演示 Kotlin 代码的简洁性，举了一个例子“按钮对象.setOnClickListener { 点击事件的处理代码 }”，这种写法正是采取了 Lamba 表达式，直接把点击事件接口的唯一方法 onClick 给省略掉。因此，本节的任务实例也可以使用类似的写法，只要说明该实例是 Runnable 类型，多余的 run 方法就能如愿去除。

下面是将任务实例按照简化形式改写后的 Kotlin 代码：

```
//把类的继承与方法重载步骤给简化掉了
private val counter = Runnable {
    count++
    tv_result.text = "当前计数值为：$count"
}
```

显而易见，上述的 counter 仍是 Runnable 类型，于是处理器依旧运行该实例即可启动任务：

```
    handler.post(counter)
```

不过这种去掉 run 方法的写法是有代价的，虽然表面上代码变得简洁，但是并不拥有类的完整结构，其内部的 this 关键字不再表示任务类自身，而是表示宿主类（即 Activity 活动类）。鉴于这点变化，该方式内部不可再调用处理器的 post 或者 postDelayed 方法，意味着此时任务实例无法重复调用自身。因此，采取简化类实例的任务对象适用于不需要重复刷新的场合。

4. 匿名实例

注意到前面第三种方式的 counter 是一个经过等号赋值的任务实例，既然这样，不如直接把等号右边的表达式加入 post 方法中，就像下面的 Kotlin 代码那样：

```
//第 1 种写法：在 post 方法中直接填写 Runnable 对象的定义代码
handler.post(Runnable {
    count++
    tv_result.text = "当前计数值为：$count"
})
```

上面的代码还可以进一步精简，因为 post 方法只能输入 Runnable 类型的参数，所以括号内部的 Runnable 纯属多余；另外，post 方法有且仅有一个输入参数，于是圆括号嵌套大括号稍显烦琐。把这两个冗余之处分别予以删除与合并，于是得到了下面匿名实例版的 Kotlin 代码：

```
//第 2 种写法：如果该任务只需执行一次，就可以采用匿名实例的方式直接嵌入任务的执行代码
handler.post {
    count++
    tv_result.text = "当前计数值为：$count"
}
```

上述去掉圆括号的办法只适合 post 方法这种仅有一个参数的调用，如果其他方法存在多个输入参数（如 postDelayed 方法），那么外层的圆括号仍需予以保留，此时大括号及其内部代码就作为一个函数参数传入。恢复圆括号的 Kotlin 调用代码如下所示：

```
//第 3 种写法：如果是延迟执行任务，就可将匿名实例作为 postDelayed 的输入参数
handler.postDelayed({
    count++
    tv_result.text = "当前计数值为：$count"
}, 1000)
```

匿名实例方式直接把任务代码写在调用函数之中，意味着这段任务代码无法被其他地方调用，所以它的适用场景更加狭小。简化类实例虽然无法重复调用自身，但是尚且允许在不同地方多次调用，而匿名实例只能在它待过的地方昙花一现，因此还是要根据实际的业务要求来选择合适的任务方式。

9.2.2　进度条 ProgressBar

本节的简单动画准备拿进度条动画练练手，因此接下来先了解一下 Android 的进度条控件 ProgressBar。进度条有两种，分别是水平进度条和圆圈进度条。水平进度条为水平方向上的一根灰色线条，允许指定最大进度和当前进度。要想在布局文件中声明水平进度条，可添加 style 属性值为 progressBarStyleHorizontal 的进度条 ProgressBar 节点，举例如下：

```
<ProgressBar
    android:id="@+id/pb_progress"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="30dp" />
```

圆圈进度条则为一个在不停转动的灰色圆圈，无法指定最大进度和当前进度。要想在布局文件中声明圆圈进度条，可添加 style 属性值为 progressBarStyle 的进度条 ProgressBar 节点，举例如下：

```
    <ProgressBar
        android:id="@+id/pb_progress"
        style="?android:attr/progressBarStyle"
        android:layout_width="match_parent"
        android:layout_height="30dp" />
```

因为圆圈进度条无法设置最大进度和当前进度，造成它的实用性不强，所以本小节主要介绍水平进度条。水平进度条的常用方法/属性在 Kotlin 与 Java 中的调用方式见表 9-1。

表 9-1　水平进度条的方法/属性在 Kotlin 与 Java 中的调用方式对比

水平进度条的方法/属性说明	Kotlin 的属性名称	Java 的方法名称
设置当前进度	progress	setProgress
设置进度条的最大值	max	setMax
设置进度条的进度图形	progressDrawable	setProgressDrawable

由于系统自带的水平进度条只是一根灰色粗线条，缺少变化，也不美观，因此实际开发中常常需要自定义进度条的样式图案。此时就得通过进度条控件的 progressDrawable 属性来设置该进度条的进度图形，注意这个进度图形不能用普通图形，只能用层次图形 LayerDrawable。层次图形可在 xml 文件中定义，倘若用于描述进度图形，则要同时定义两个层次，即背景层次与进度条层次。

下面是一个层次图形定义的 xml 例子，其中根节点 layer-list 表示这是一个层次列表，即层次图形定义。其下面再定义两个层次，其中背景层次的 id 为@android:id/background，采用的是形状图形（节点名称为 shape）；进度条层次的 id 为@android:id/progress，采用的是裁剪图形 ClipDrawable（节点名称为 clip）：

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@android:id/background">
        <shape>
            <solid android:color="#333333" />
        </shape>
    </item>
    <item android:id="@android:id/progress">
        <clip>
            <nine-patch android:src="@drawable/notify_green" />
        </clip>
    </item>
</layer-list>
```

在 Activity 中使用进度条控件的 Kotlin 代码如下所示，主要是根据输入的进度数值展示进度条的当前进度：

```
class ProgressBarActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_progress_bar)
```

```
            //设置最大进度
            pb_progress.max = 100
            //设置默认进度
            pb_progress.progress = 0
            //设置进度条图形
            pb_progress.progressDrawable = resources.getDrawable
(R.drawable.progress_green)
            btn_progress.setOnClickListener {
                //根据输入的进度数值展示进度条的当前进度
                pb_progress.progress = et_progress.text.toString().toInt()
            }
        }
    }
```

进度条的进度数值变更前后的界面效果如图 9-7 与图 9-8 所示，其中图 9-7 所示为进度值为 0 的界面，此时只有一根黑色的进度条背景；图 9-8 所示为进度值为 60 的界面，此时绿色进度条占据全部进度的 60%长度。



图 9-7　当前进度为 0 的进度条界面

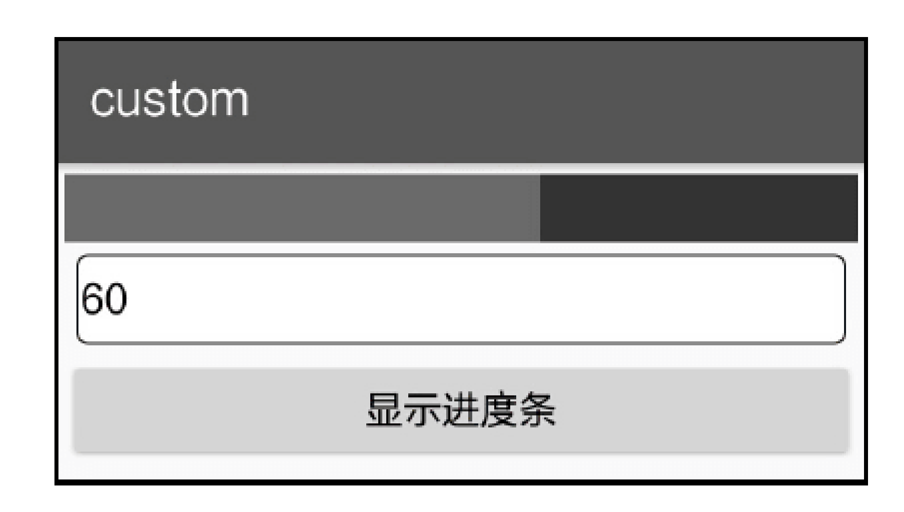


图 9-8　当前进度为 60 的进度条界面

9.2.3　自定义文本进度条

不过，就算进度条用上了自定义的进度图形，仍然存在不足之处。比如进度条仅仅显示一段进度图形，并未显示进度数值的文本，用户如何才能得知当前的具体进度？故而可在进度条中间增加文字提示，使之符合用户的视觉需求。但是 ProgressBar 没有提供设置文本的方法，于是要想在进度条中央显示进度文字，就得基于 ProgressBar 自定义一个进度条。自定义进度条的思路主要是重写 onDraw 绘图函数，在该函数中调用 canvas 的 drawText 方法，往进度条的中央位置添加进度文本。

下面是上述文本进度条的 Kotlin 自定义代码例子：

```
    //自定义视图要在类名后面增加“@JvmOverloads constructor”，因为布局文件中的自定义视图必须兼容 Java
    class TextProgressBar @JvmOverloads constructor(context: Context, attrs:
AttributeSet? = null, defStyle: Int = 0) : ProgressBar(context, attrs, defStyle) {
        var progressText = ""
        private var paint: Paint
        private var textColor = Color.WHITE
```

```
    private var textSize = 30f

    init {
        //初始化画笔
        paint = Paint()
        paint.color = textColor
        paint.textSize = textSize
    }

    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        val rect = Rect()
        //获得进度文本的矩形边界
        paint.getTextBounds(progressText, 0, progressText.length, rect)
        val x = width / 2 - rect.centerX()
        val y = height / 2 - rect.centerY()
        //把文本内容绘制在进度条的中间位置
        canvas.drawText(progressText, x.toFloat(), y.toFloat(), paint)
    }
}
```

自定义文本进度条的显示效果如图 9-9 与图 9-10 所示，其中图 9-9 展示当前进度为 40%时的界面，图 9-10 展示当前进度为 80%时的界面。



图 9-9　当前进度为 40%的文本进度条



图 9-10　当前进度为 80%的文本进度条

9.2.4　实现进度条动画

现在有了任务 Runnable 和自定义的文本进度条 TextProgressBar 做铺垫，就能很方便地实现进度条滚动刷新的动画效果了。先把“9.2.1　任务 Runnable”小节提到的计数器 Runnable 代码拿过来，然后补充对进度值的判断处理：在当前进度未达到 100%时，更新文本进度条的进度值与进度文本；一旦当前进度超过 100%，就停止进度更新，并展示 100%进度时的控件界面。

据此编写进度条动画的 Kotlin 页面代码如下所示：

```
class ProgressAnimationActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_progress_animation)
        btn_animation.setOnClickListener {
```

```
            btn_animation.isEnabled = false
            //延迟 50 毫秒开始进度条动画
            handler.postDelayed(animation, 50)
        }
    }

    private var mProgress = 0
    private val handler = Handler()
    //定义一个刷新进度条的任务
    private val animation = object : Runnable {
        override fun run() {
            if (mProgress <= 100) {
                tpb_progress.progress = mProgress
                tpb_progress.progressText = "当前处理进度为$mProgress%"
                //当前进度未满 100%，继续进度刷新动画
                handler.postDelayed(this, 50)
                mProgress++
            } else {
                //进度条动画结束，恢复初始进度数值
                mProgress = 0
                btn_animation.isEnabled = true
            }
        }
    }
}
```

接下来观看一下进度条动画的演示结果，如图 9-11 所示，此时进度条动画播放到了 51%；如图 9-12 所示，此时进度条动画播放完毕，当前进度停留在 100%的位置。



图 9-11　进度条动画正在播放



图 9-12　进度条动画结束播放

9.3　自定义通知栏

通知栏 Notification 是 Android 倾力打造的一个重要部件，几乎每个大版本都对通知栏进行了卓有成效的升级，那么最新的通知栏究竟具备哪些激动人心的功能？Kotlin 又是怎样编码实现推送通知的？带着这些疑问，本节将一步一步抽丝剥茧，把通知栏的各个重要特性及其用法逐步地展现在读者面前。

9.3.1 通知推送 Notification

在手机屏幕的顶端下拉会弹出一列通知栏，里面放的是各个 App 想即时提醒用户的消息，这个消息内容是由系统的通知服务产生并推送的。每条消息通知基本都有这些元素：图标、标题、内容、时间等，它的消息参数通过 Notification.Builder 构建，下面介绍常用的消息参数构建方法。

- setAutoCancel：设置该通知是否自动清除。若为 true，则点击该通知后，通知会自动消失；若为 false，则点击该通知后，通知不会消失。
- setWhen：设置推送时间，格式为“小时：分钟”。推送时间在通知栏右方显示。
- setShowWhen：设置是否显示推送时间。
- setUsesChronometer：设置是否显示计数器。为 true 时将不显示推送时间，动态显示从通知被推送到当前的时间间隔，以“分钟：秒钟”格式显示。
- setSmallIcon：设置状态栏里面的图标（小图标）。
- setTicker：设置状态栏里面的提示文本。
- setLargeIcon ：设置通知栏里面的图标（大图标）。
- setContentTitle：设置通知栏里面的标题文本。
- setContentText：设置通知栏里面的内容文本。
- setSubText：设置通知栏里面的附加说明文本，位于内容文本下方。若调用该方法，则 setProgress 的设置将失效。
- setProgress：设置进度条与当前进度。进度条位于标题文本与内容文本中间。
- setNumber：设置通知栏右下方的数字，可与 setProgress 联合使用，表示当前的进度数值。
- setContentInfo：设置通知栏右下方的文本。若调用该方法，则 setNumber 的设置将失效。
- setContentIntent：设置内容的延迟意图 PendingIntent，在点击该通知时触发该意图。通常调用 PendingIntent 的 getActivity 方法获得延迟意图对象，getActivity 表示点击后跳转到该页面。
- setDeleteIntent：设置删除的延迟意图 PendingIntent，在滑掉该通知时触发该动作。
- build：构建方法。在以上参数都设置完毕后，调用该方法会返回 Notification 对象。

接下来演示两个不同样式的通知消息：静止的简单消息和动态的计时消息。

1. 静止的简单消息

大多数消息通知都是这种静态的文本消息，通常简简单单一句话告诉用户有什么新鲜事了、推出什么新活动了等。然后待用户点击该通知后，就跳到设定好的活动页面。构建简单消息的时候，需要注意下面两点：

（1）setSmallIcon 方法必须调用，否则不会显示通知消息。

（2）setSubText 与 setProgress 两个方法同时只能调用其一，因为附加说明与进度条都位于标题文本的下方。

下面是发送简单消息的 Kotlin 代码片段：

```
        private fun sendSimpleNotify(title: String, message: String) {
            //声明一个点击通知消息时触发的动作意图
            val clickIntent = intentFor<MainActivity>()
            val piClick = PendingIntent.getActivity(this,
                    R.string.app_name, clickIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //开始构建简单消息的各个参数
            val builder = Notification.Builder(this)
            val notify = builder.setContentIntent(piClick)
                    .setAutoCancel(true)
                    .setSmallIcon(R.drawable.ic_app)
                    .setSubText("这里是副本")
                    .setTicker("简单消息来啦")
                    .setWhen(System.currentTimeMillis())
                    .setLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.ic_app))
                    .setContentTitle(title)
                    .setContentText(message).build()
            //获取系统的通知管理器
            val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
            notifyMgr.notify(R.string.app_name, notify)
        }
```

简单消息的通知栏效果如图 9-13 所示，可见通知栏左边是图标，中间是标题与内容，右边是推送时间。



图 9-13　简单消息的通知栏效果

2. 动态的计时消息

在消息通知的右边放置一个计时器 Chronometer，这便形成了能够动态显示已逝去时间的计时消息，该消息常用于度量某个特殊事务的耗时时长。构建计时消息的时候，需要注意下面两点：

（1）setWhen 与 setUsesChronometer 两个方法同时只能调用其一，也就是说推送时间与计数器无法同时显示，因为它们都位于通知栏的右边。

（2）setNumber 与 setContentInfo 两个方法同时只能调用其一，因为计数值与提示都位于通知栏的右下方。

下面是发送计时消息的 Kotlin 代码片段：

```
        private fun sendCounterNotify(title: String, message: String) {
            //声明一个删除通知消息时触发的动作意图
            val cancelIntent = intentFor<MainActivity>()
```

```
            val piDelete = PendingIntent.getActivity(this,
                    R.string.app_name, cancelIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //开始构建计时消息的各个参数
            val builder = Notification.Builder(this)
            val notify = builder.setDeleteIntent(piDelete)
                    .setAutoCancel(true)
                    .setUsesChronometer(true)
                    .setProgress(100, 60, false)
                    .setNumber(99)
                    .setSmallIcon(R.drawable.ic_app)
                    .setTicker("计数消息来啦")
                    .setLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.ic_app))
                    .setContentTitle(title)
                    .setContentText(message).build()
            //获取系统的通知管理器
            val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
            notifyMgr.notify(R.string.app_name, notify)
        }
```

计时消息的通知栏效果如图 9-14 所示，可见通知栏左边是图标，中间是标题文本、进度条、内容文本，右边是计时器与计数值。



图 9-14　计时消息的通知栏效果

9.3.2　大视图通知

9.3.1 小节介绍了消息通知的基本样式，从效果图看到每条通知都只有窄窄的一行消息，而且交互方式也比较简陋，未免显得小家子气。于是 Android 从 API 16 开始（Android 4.1）引入了新的大视图通知，所谓大视图通知，就是块头大（即高度增加）了。除了高度增加外，大视图通知也新增了几项好用的功能，比如允许设置通知到达的提醒方式、允许设置多个按钮分别指定响应方式等，如此一来，不但块头变大，功能也增强了，名副其实地做到了大气磅礴。

大视图通知的参数出人意料地没有继续使用 Notification.Builder 构建，而是采用新来的 NotificationCompat.Builder。当然，新来的伙计除了掌握 Notification.Builder 已有的参数设置方法外，另外还添加了几个大视图额外功能的对应方法，这几个新方法的功能具体说明如下。

- setDefaults：设置通知到达的提醒方式。可同时设置多种通知方式，每种通知方式之间使用位运算符 “or” 连接。提醒方式的取值说明见表 9-2。

表 9-2　通知到达的提醒方式取值说明

通知到达的提醒方式	说明
Notification.DEFAULT_ALL	默认所有
Notification.DEFAULT_SOUND	默认铃声
Notification.DEFAULT_VIBRATE	默认震动
Notification.DEFAULT_LIGHTS	默认闪光

- addAction：在本通知底部添加一个动作按钮，可同时添加多个按钮，这些按钮从左到右排列。该方法的第一个参数为按钮图片的资源 ID，第二个参数为按钮的文本内容，第三个参数为点击按钮对应的动作意图。
- setStyle：设置该通知的大视图风格。大视图通知有三种风格类型，分别是大文本风格 NotificationCompat.BigTextStyle、大图片风格 NotificationCompat.BigPictureStyle 和收件箱风格 NotificationCompat.InboxStyle。具体的风格类型取值说明见表 9-3。

表 9-3　大视图通知的三种风格类型说明

大视图说明	大视图风格	内容设置方法	方法的说明
大文本风格	BigTextStyle	bigText	设置大视图中间的文本内容
大图片风格	BigPictureStyle	bigPicture	设置大视图中间的图片内容
收件箱风格	InboxStyle	addLine	添加一行消息文本。可多次调用该方法，每调用一次就添加一行消息

除了表 9-3 列出的特有方法外，三种大视图风格还共同支持下面的两种风格设置方法。

- setBigContentTitle：设置大视图通知的标题文本。
- setSummaryText：设置大视图通知的摘要文本。摘要文本位于底部按钮的上方。

需要注意的是，大视图通知并不总是完全展开显示，大多数情况仍然像一般通知那样只显示狭窄的一行，只有在该通知处于通知栏顶部并且用户将其下拉时，大视图通知才会向下展示完整的大篇幅消息。

下面是构建并发送大视图通知的 Kotlin 代码片段：

```
    private fun getStyleAndSend(title: String, message: String, type: Int) {
        var style: NotificationCompat.Style? = null
        when (type) {
            0 -> { //声明大文本风格
                style = NotificationCompat.BigTextStyle()
                style.setBigContentTitle(title)
                style.setSummaryText(message)
                style.bigText("这是一条大文字风格的通知消息")
            }
            1 -> { //声明大图片风格
                style = NotificationCompat.BigPictureStyle()
```

```
                style.setBigContentTitle(title)
                style.setSummaryText(message)
                style.bigLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.icon_financer))
                style.bigPicture(BitmapFactory.decodeResource(resources,
R.drawable.icon_sunshine))
            }
            2 -> { //声明收件箱风格
                style = NotificationCompat.InboxStyle()
                style.setBigContentTitle(title)
                style.setSummaryText(message)
                style.addLine("天青色等烟雨，而我在等你")
                style.addLine("炊烟袅袅升起，隔江千万里")
                style.addLine("在瓶底书汉隶仿前朝的飘逸")
            }
        }
        sendLargeNotify(title, message, style)
        toast("大视图消息已推送到通知栏。")
    }

    private fun sendLargeNotify(title: String, message: String, style:
NotificationCompat.Style?) {
        //声明一个“取消”按钮的动作意图
        val cancelIntent = intentFor<NotifyLargeActivity>()
        val piCancel = PendingIntent.getActivity(this,
                R.string.app_name, cancelIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        //声明一个“前往”按钮的动作意图
        val confirmIntent = intentFor<NotifyLargeActivity>()
        val piConfirm = PendingIntent.getActivity(this,
                R.string.app_name, confirmIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        //大视图通知需要通过 NotificationCompat.Builder 来构建
        val builder = NotificationCompat.Builder(this)
        builder.setSmallIcon(R.drawable.ic_app)
                .setTicker("大视图消息来啦")
                .setWhen(System.currentTimeMillis())
                .setLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.ic_app))
                .setContentTitle(title)
                .setContentText(message)
                .setDefaults(Notification.DEFAULT_ALL) //设置大视图通知的提醒方式
                .setStyle(style) //设置大视图通知的风格类型
                .addAction(R.drawable.icon_cancel, "取消", piCancel) //添加取消
按钮及其动作
```

```
                .addAction(R.drawable.icon_confirm, "前往", piConfirm) //添加前
往按钮及其动作
        val notify = builder.build()
        //获取系统的通知管理器
        val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
        notifyMgr.notify(1, notify)
    }
```

大视图通知的推送效果如图 9-15～图 9-17 所示，其中图 9-15 展示大文本风格的通知栏界面，图 9-16 展示大图片风格的通知栏界面，图 9-17 展示收件箱风格的通知栏界面。



图 9-15　大文本风格的通知栏界面



图 9-16　大图片风格的通知栏界面



图 9-17　收件箱风格的通知栏界面

9.3.3　三种特殊的通知类型

除了像大视图通知那样拓展显示高度外，消息通知还有其他的几种特殊显示方式，包括进度通知、浮动通知、锁屏通知等，分别介绍如下。

1. 进度通知

进度通知指的是在通知栏动态刷新进度的消息通知，前面“9.3.1 通知推送 Notification”小节提到可以通过 setProgress 方法设置进度条与当前进度，但是该方法调用之后仅仅展示静态的进度条，要想让进度值持续前进，得利用延时任务不断刷新最新的进度，有关延时任务的说明参见“9.2.1 任务 Runnable”小节。

刷新通知其实很简单，只要反复发送相同标识的消息到通知栏即可，下面是演示进度通知的 Kotlin 代码片段：

```
    private val handler = Handler()
    private var count = 0
    private var refreshNotify: ProgressNotify? = null
    //开始播放进度通知的刷新动画
    private fun startProgressNotify(title: String, message: String) {
        count = 0
        refreshNotify = ProgressNotify(title, message)
        handler.post(refreshNotify)
        toast("已推送到进度通知。")
    }
```

```
        //定义一个持续发送进度通知的任务内部类，当进度超过 100%时任务停止
        private inner class ProgressNotify(private val title: String, private val
message: String) : Runnable {
            override fun run() {
                sendProgressNotify(title, message, count)
                count++
                if (count <= 100) {
                    //若当前进度没有超过 100%，则继续刷新通知栏上的进度
                    handler.postDelayed(refreshNotify, 200)
                }
            }
        }

        //发送单次进度通知。若要不断刷新进度，则需外部多次调用该方法
        private fun sendProgressNotify(title: String, message: String, progress:
Int) {
            val clickIntent = intentFor<MainActivity>()
            val piClick = PendingIntent.getActivity(this,
                    R.string.app_name, clickIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //开始构建进度通知的各个参数
            val builder = Notification.Builder(this)
            builder.setContentIntent(piClick)
                    .setAutoCancel(true)
                    .setSmallIcon(R.drawable.ic_app)
                    .setTicker("进度通知来啦")
                    .setWhen(System.currentTimeMillis())
                    .setLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.ic_app))
                    .setContentTitle(title)
                    .setContentText(message)
                    .setProgress(100, progress, false) //设置进度条的当前进度
            val notify = builder.build()
            //获取系统的通知管理器
            val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
            notifyMgr.notify(R.string.app_name, notify)
        }
```

进度通知的播放效果如图 9-18 和图 9-19 所示，其中图 9-18 所示为开始播放进度通知的通知栏界面，图 9-19 所示为结束播放进度通知的通知栏界面。

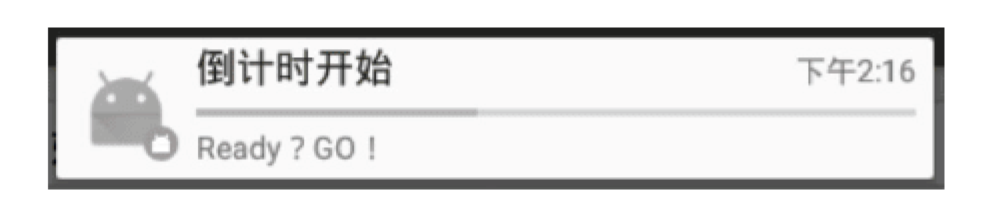


图 9-18　进度通知开始播放

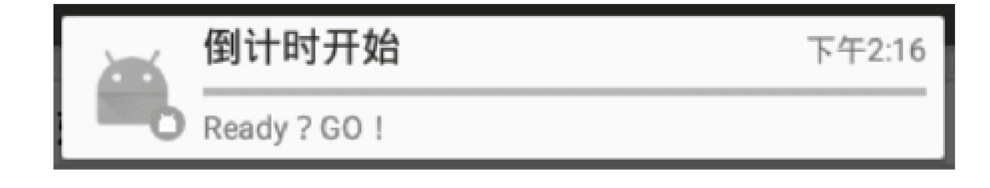


图 9-19　进度通知结束播放

2. 浮动通知

浮动通知指的是不离开当前页面并在屏幕顶部悬挂显示的消息通知，该功能需要 Android 5.0 及更高版本的系统支持。浮动通知的应用场景挺广的，包括但不限于收到新的短信、微信群有人发红包等。设置浮动通知的关键是调用 Notification.Builder 对象的 setFullScreenIntent 方法，该方法指定了浮动窗的点击事件以及优先级。

下面是演示浮动通知的 Kotlin 代码片段：

```
    private fun sendFloatNotify(title: String, message: String) {
        toast("已推送到浮动通知。")
        val clickIntent = intentFor<MainActivity>()
        val piClick = PendingIntent.getActivity(this,
                R.string.app_name, clickIntent, PendingIntent.
FLAG_UPDATE_CURRENT)
        //开始构建浮动通知的各个参数
        val builder = Notification.Builder(this)
        builder.setContentIntent(piClick)
                .setAutoCancel(true)
                .setSmallIcon(R.drawable.ic_app)
                .setTicker("浮动通知来啦")
                .setWhen(System.currentTimeMillis())
                .setLargeIcon(BitmapFactory.decodeResource(resources,
R.drawable.ic_app))
                .setContentTitle(title)
                .setContentText(message)
                .setFullScreenIntent(piClick, true) //设置浮动窗的点击事件以及优先级
        val notify = builder.build()
        //获取系统的通知管理器
        val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
        notifyMgr.notify(R.string.app_name, notify)
    }
```

浮动通知的推送效果如图 9-20 所示，此时屏幕顶端出现了一条悬浮着的通知消息，点击该消息可执行对应的页面跳转动作。



图 9-20　浮动通知的推送效果

3. 锁屏通知

锁屏通知指的是在锁屏界面依然提示通知内容的消息通知，该功能需要 Android 5.0 及更高版本的系统支持。设置锁屏通知的关键是调用 Notification.Builder 对象的 setVisibility 方法，该方法用于指定通知消息在锁屏状态下的显示方式，显示方式的取值说明见表 9-4。

表 9-4　锁屏通知在锁屏状态下的显示方式取值说明

Notification 类的锁屏显示方式	说明
Notification.VISIBILITY_PRIVATE	显示基本信息，如通知的图标，但隐藏通知的全部内容
Notification.VISIBILITY_PUBLIC	显示通知的全部内容
Notification.VISIBILITY_SECRET	不显示任何内容，包括图标

下面是演示锁屏通知的 Kotlin 代码片段：

```
    private fun sendLockNotify(title: String, message: String, visibile: Boolean) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            toast("锁屏通知需要 5.0 以上系统支持。")
            return
        } else {
            toast("已推送锁屏通知。")
        }
        val clickIntent = intentFor<MainActivity>()
        val piClick = PendingIntent.getActivity(this,
                R.string.app_name, clickIntent, PendingIntent.FLAG_UPDATE_CURRENT)
        //开始构建锁屏通知的各个参数
        val builder = Notification.Builder(this)
        builder.setContentIntent(piClick)
                .setAutoCancel(true)
                .setSmallIcon(R.drawable.ic_app)
                .setTicker("锁屏通知来啦")
                .setWhen(System.currentTimeMillis())
                .setLargeIcon(BitmapFactory.decodeResource(resources, R.drawable.ic_app))
                .setContentTitle(title)
                .setContentText(message)
                //设置锁屏通知的可见性
                .setVisibility(if (visibile) Notification.VISIBILITY_PUBLIC else Notification.VISIBILITY_PRIVATE)
        val notify = builder.build()
        //获取系统的通知管理器
        val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notifyMgr.notify(R.string.app_name, notify)
    }
```

锁屏通知的推送效果如图 9-21 所示，此时系统进入锁屏状态，并且锁屏界面仍旧展示着锁屏通知的消息内容。



图 9-21　锁屏通知的推送效果

9.3.4　远程视图 RemoteViews

前两个小节介绍了消息通知的非常规展现形式，但基本没涉及通知内部的布局格式。我们知道页面可以自己定义布局，碎片也可以自己定义布局，乃至单个列表项都能自己定义布局，那么消息通知能否也自定义布局呢？这是肯定的，不过要借助于远程视图 RemoteViews 方能实现。其实 Notification.Builder 已经提供了 setContent 方法，该方法就是用来设置当前通知的 RemoteViews 对象，一旦调用了 setContent 方法，RemoteViews 对象指定的自定义布局就会替换掉系统默认的通知栏布局。

与活动页面相比，远程视图是一个不但小型而且简化了的页面，简化的意思是功能减少了，限制变多了。虽然 RemoteViews 与 Activity 一样都有自己的布局文件，但是 RemoteViews 的使用权限大幅缩小，二者之间的区别主要有：

（1）RemoteViews 主要用于通知栏部件和桌面部件的布局，而 Activity 用于活动页面的布局。

（2）RemoteViews 只支持少数几种控件，如 TextView、ImageView、Button、ImageButton、ProgressBar、Chronometer（计时器）、AnalogClock（模拟时钟）。

（3）RemoteViews 不可直接获取和设置控件信息，只能通过该对象的 set 方法来修改控件信息。

下面是远程视图 RemoteViews 的常用方法。

- 构造函数：创建一个 RemoteViews 对象。第一个参数是包名，第二个参数是布局文件 id。
- setViewVisibility：设置指定控件是否可见。
- setViewPadding：设置指定控件的间距。
- setTextViewText：设置指定 TextView 或 Button 控件的文字内容。
- setTextViewTextSize：设置指定 TextView 或 Button 控件的文字大小。
- setTextColor：设置指定 TextView 或 Button 控件的文字颜色。
- setTextViewCompoundDrawables：设置指定 TextView 或 Button 控件的文字周围图标。
- setImageViewResource：设置 ImageView 或 ImgaeButton 控件的资源编号。
- setImageViewBitmap：设置 ImageView 或 ImgaeButton 控件的位图对象。
- setChronometer：设置计时器信息。
- setProgressBar：设置进度条信息，包括最大值与当前进度。
- setOnClickPendingIntent：设置指定控件的点击响应动作。

完成 RemoteViews 对象的构建与设置之后，再调用 Notification.Builder 对象的 setContent 方法，即可完成自定义通知布局的设置。下面是一个远程视图用到的播放器通知布局文件例子：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:minHeight="64dp"
    android:orientation="horizontal" >

    <ImageView
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:scaleType="fitCenter"
        android:src="@drawable/tt" />

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_marginLeft="3dp"
        android:layout_marginRight="3dp"
        android:layout_weight="4"
        android:orientation="vertical" >

        <ProgressBar
            android:id="@+id/pb_play"
            style="?android:attr/progressBarStyleHorizontal"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:max="100"
            android:progress="10" />

        <TextView
            android:id="@+id/tv_play"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:textColor="#ffffff"
            android:textSize="17sp" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:orientation="vertical" >

        <Chronometer
            android:id="@+id/chr_play"
```

```
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:gravity="center" />

        <Button
            android:id="@+id/btn_play"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="2"
            android:text="暂停"
            android:textColor="#ffffff"
            android:textSize="15sp" />
    </LinearLayout>
</LinearLayout>
```

下面是获取播放器通知布局的 Kotlin 代码片段：

```
    //获取播放器的通知栏布局
    private fun getNotifyMusic(ctx: Context, event: String, song: String, isPlay:
Boolean, progress: Int, time: Long): RemoteViews {
        //从 notify_music.xml 布局文件构造远程视图对象
        val notify_music = RemoteViews(ctx.packageName, R.layout.notify_music)
        if (isPlay) {
            notify_music.setTextViewText(R.id.btn_play, "暂停")
            notify_music.setTextViewText(R.id.tv_play, song + "正在播放")
            notify_music.setChronometer(R.id.chr_play, time, "%s", true)
        } else {
            notify_music.setTextViewText(R.id.btn_play, "继续")
            notify_music.setTextViewText(R.id.tv_play, song + "暂停播放")
            notify_music.setChronometer(R.id.chr_play, time, "%s", false)
        }
        notify_music.setProgressBar(R.id.pb_play, 100, progress, false)
        val pIntent = Intent(event)
        val piPause = PendingIntent.getBroadcast(
                ctx, R.string.app_name, pIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        //设置暂停/继续按钮的点击动作对应的广播事件
        notify_music.setOnClickPendingIntent(R.id.btn_play, piPause)
        return notify_music
    }
```

自定义通知栏的演示效果如图 9-22 所示，可以看到播放器图标在通知栏左边，进度条在上方，歌曲名称在下方，计时器与控制按钮分布在通知栏右边。

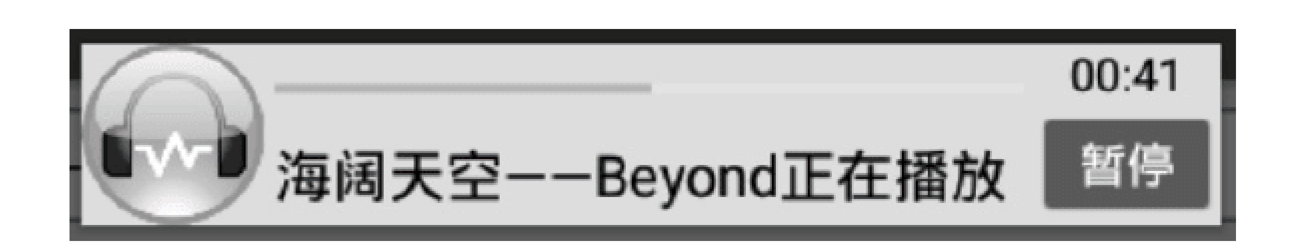


图 9-22　自定义通知栏的演示效果

9.3.5　自定义折叠式通知

前面在“9.3.2 大视图通知”小节提到 Android 从 4.1 之后允许推送大视图通知，其实这个大视图的布局界面也是能够自定义的，当然依旧需要借助远程视图 RemoteViews 的力量。这个自定义的大视图通知可以看作是一种折叠式通知，在它处于折叠状态时显示普通样式的通知内容，在它处于展开状态时显示自定义的大视图格式内容。自定义普通视图与自定义大视图的设置方式参见表 9-5 的说明。

表 9-5　自定义普通视图与自定义大视图的设置方式

视图设置方式说明	API24 以下的设置方式	API24 及以上的设置方式 Notification.Builder 的方法
指定自定义的普通视图	Notification.Builder 的 setContent 方法	setCustomContentView
指定自定义的普通视图	Notification 的 contentView 属性	setCustomContentView
指定自定义的展开视图	Notification 的 bigContentView 属性	setCustomBigContentView

下面是获取播放器通知的大视图布局 Kotlin 代码片段：

```
    private fun sendCustomNotify(ctx: Context, song: String, contentView: RemoteViews, bigContentView: RemoteViews?) {
        //声明一个点击通知消息时触发的动作意图
        val intent = ctx.intentFor<MainActivity>()
        val contentIntent = PendingIntent.getActivity(ctx,
                R.string.app_name, intent, PendingIntent.FLAG_UPDATE_CURRENT)
        val builder = Notification.Builder(ctx)
        builder.setContentIntent(contentIntent)
                .setContent(contentView) //采用自定义的通知布局
                .setTicker(song)
                .setSmallIcon(R.drawable.tt_s)
        val notify = builder.build()
        //正常高度的自定义通知
        notify.contentView = contentView
        if (bigContentView != null) {
            //展开后的自定义通知
            notify.bigContentView = bigContentView
        }
        //获取系统的通知管理器
        val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
```

```
            notifyMgr.notify(R.string.app_name, notify)
        }
        //获取折叠视图展开后的通知栏布局
        private fun getNotifyExpand(ctx: Context, event: String, song: String,
isPlay: Boolean, progress: Int, time: Long): RemoteViews {
            //从 notify_expand.xml 布局文件构造远程视图对象
            val notify_expand = RemoteViews(ctx.packageName,
R.layout.notify_expand)
            if (isPlay) {
                notify_expand.setTextViewText(R.id.btn_play, "暂停")
                notify_expand.setTextViewText(R.id.tv_play, song + "正在播放")
                notify_expand.setChronometer(R.id.chr_play, time, "%s", true)
            } else {
                notify_expand.setTextViewText(R.id.btn_play, "继续")
                notify_expand.setTextViewText(R.id.tv_play, song + "暂停播放")
                notify_expand.setChronometer(R.id.chr_play, time, "%s", false)
            }
            notify_expand.setProgressBar(R.id.pb_play, 100, progress, false)
            val pIntent = Intent(event)
            val piPause = PendingIntent.getBroadcast(
                    ctx, R.string.app_name, pIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //设置播放按钮的点击动作对应的广播事件
            notify_expand.setOnClickPendingIntent(R.id.btn_play, piPause)
            val bIntent = Intent(ctx, NotifyCustomActivity::class.java)
            val piBack = PendingIntent.getActivity(ctx,
                    R.string.app_name, bIntent, PendingIntent.FLAG_UPDATE_CURRENT)
            //设置返回按钮的点击动作对应的跳转事件
            notify_expand.setOnClickPendingIntent(R.id.btn_back, piBack)
            return notify_expand
        }
```

自定义折叠式通知的演示效果如图 9-23 和图 9-24 所示，其中图 9-23 所示为展开状态时的通知栏界面，图 9-24 所示为折叠状态时的通知栏界面。



图 9-23　自定义折叠式通知的展开效果

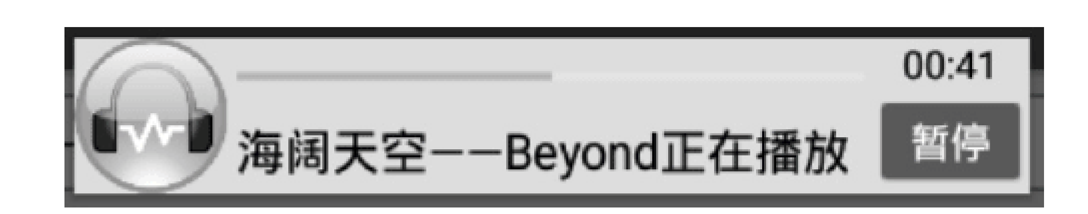


图 9-24　自定义折叠式通知的折叠效果

9.4 Service 服务启停

服务 Service 是 Android 的四大组件之一，常用在不见其人、但闻其声的隐秘场合，像上一节的通知推送用到了系统的通知服务 NOTIFICATION_SERVICE。既然 Android 有自带的系统服务，那么 App 也可以有自己的私人服务。对于 App 自身的服务，开发者要如何控制服务的运行过程？使用 Kotlin 编码进行服务启停又有哪些值得注意的地方？本节会通过详细的讲解逐步回答这些问题。

9.4.1 普通方式启动服务

启动服务 Service 的时机各不相同，既可以在活动 Activity 中启动服务，也可以在广播接收器 Receiver 中启动服务，甚至能够在 Application 中启动服务。启动服务的普通方式很简单，使用 Java 编码不过下面两行代码而已：

```
        Intent intent = new Intent(ServiceNormalActivity.this,
NormalService.class);
        startService(intent);
```

把上面启动服务的 Java 代码直译为 Kotlin，翻译后的 Kotlin 代码如下所示：

```
        val intent = Intent(this@ServiceNormalActivity,
NormalService::class.java)
        startService(intent)
```

至少表面看起来这里的 Kotlin 编码跟 Java 编码半斤八两。不过倘若读者有心的话，定然发现第 6 章的“6.4.1 传送配对字段数据”早已提到了 Anko 库对 startActivity 的简化写法，同样 Anko 库也提供了对 startService 的简化写法，故而上述的 Kotlin 服务启动代码可以简写成下面这般：

```
        startService<NormalService>()
```

因为以上的简化写法用到了 Anko 库的扩展函数，所以必须先导入 Anko 库的指定代码，即在 kt 文件头部添加下面一行导入语句：

```
    import org.jetbrains.anko.startService
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
        compile "org.jetbrains.anko:anko-common:$anko_version"
```

启动服务的同时，允许携带请求参数传递给该服务，传送请求参数的 Kotlin 服务启动代码如下所示：

```
        startService<NormalService>("request_content" to
et_request.text.toString())
```

不过由于服务既有启动操作又有停止操作，并且停止服务的时候需要传入原来启动服务时的 Intent 对象，因此启动服务更常见的做法是：先声明一个意图对象，然后把它作为请求参数在启动服务时传入，这样停止服务时就能传入同样的意图对象。按照这个思路重新编码，于是形成了下述引进 intentFor 函数的 Kotlin 代码：

```
        val intent = intentFor<NormalService>("request_content" to
et_request.text.toString())
        startService(intent)
```

上面传输请求参数时利用关键字隔开参数名称与参数值，该写法容易混淆，此时可以采取 Pair 配对的形式传递参数，改写后的 Kotlin 启动服务代码如下所示：

```
        val intent = intentFor<NormalService>(Pair("request_content",
et_request.text.toString()))
        startService(intent)
```

学习完前面的几种服务启动写法，接着通过完整的代码演示看看 Kotlin 操纵服务的全过程，下面是演示用的 Kotlin 页面代码：

```
    class ServiceNormalActivity : AppCompatActivity() {
        var intentNormal: Intent? = null

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_service_normal)
            btn_start.setOnClickListener {
                //第一种写法，参数名和参数值使用关键字 to 隔开
                intentNormal = intentFor<NormalService>("request_content" to
et_request.text.toString())
                //第二种写法，利用 Pair 把参数名和参数值进行配对
                //intentNormal = intentFor<NormalService>(Pair("request_content",
et_request.text.toString()))
                startService(intentNormal)
                //虽然 Anko 库集成了 startService 的简化写法，但是一般不这么调用
                //因为服务启动之后是需要停止的，按 Anko 的简化写法会无法停止服务
                //除非无须在代码中停止该服务，才可以采用 Anko 的简化写法
                //startService<NormalService>("request_content" to
et_request.text.toString())
                toast("普通服务已启动")
            }
            btn_stop.setOnClickListener {
                if (intentNormal != null) {
                    stopService(intentNormal)
                    toast("普通服务已停止")
                }
            }
            Normal.tv_normal = findViewById<TextView>(R.id.tv_normal)
```

```
        }

        companion object Normal {
            private var tv_normal: TextView? = null
            private var mDesc = ""
            //静态方法 showText 给 NormalService 内部调用
            fun showText(desc: String) {
                mDesc = "${mDesc}${DateUtil.nowTime} $desc\n"
                //如果 tv_normal 非空才设置文本，否则不设置文本
                tv_normal?.text = mDesc
            }
        }
    }
```

下面是普通方式所要启动的 Kotlin 服务代码例子：

```
    class NormalService : Service() {

        override fun onCreate() {
            ServiceNormalActivity.showText("创建服务")
            super.onCreate()
        }

        override fun onStartCommand(intent: Intent, flags: Int, startid: Int):
Int {
            val bundle = intent.extras
            val request_content = bundle.getString("request_content")
            ServiceNormalActivity.showText("启动服务，收到请求内容：
${request_content}")
            return Service.START_STICKY
        }

        override fun onDestroy() {
            ServiceNormalActivity.showText("停止服务")
            super.onDestroy()
        }

        override fun onBind(intent: Intent): IBinder? = null
    }
```

然后运行测试应用，呈现出来的服务启停效果如图 9-25 和图 9-26 所示，其中图 9-25 所示为启动服务的界面，图 9-26 所为停止服务的界面。



图 9-25　服务启动的效果

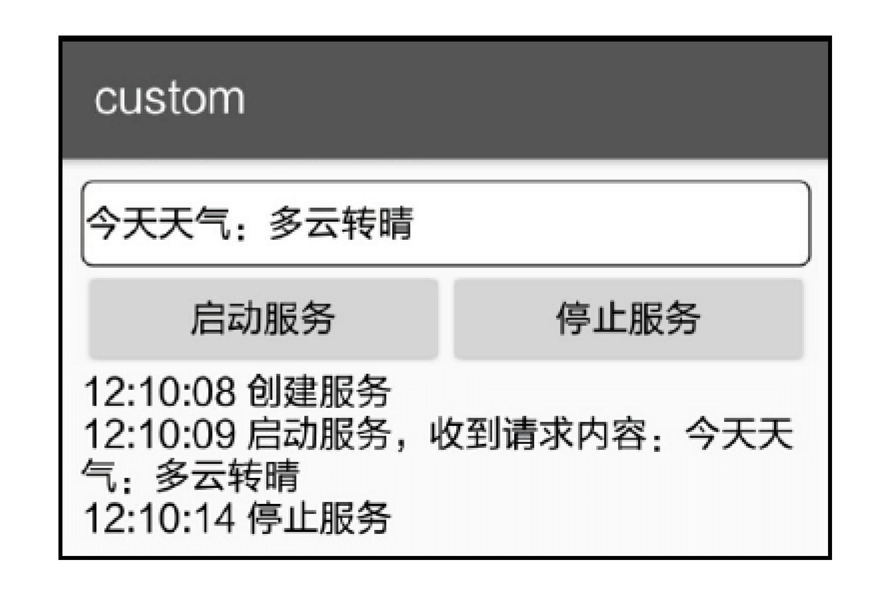


图 9-26　服务停止的效果

9.4.2　绑定方式启动服务

9.4.1 小节讲到了使用普通方式启停服务，其实启停服务还有另一种绑定方式，调用 bindService 方法表示绑定并启动服务（如果原来没有启动），调用 unbindService 方法表示解除绑定并停止服务（如果原来没有启动）。此时一样能够利用 intentFor 扩展函数来构建意图对象，其余的代码流程则基本跟 Java 保持一致，下面是以绑定方式启动服务的 Kotlin 代码例子：

```
    class ServiceBindActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_service_bind)
            Bind.tv_bind = findViewById<TextView>(R.id.tv_bind)
            btn_start_bind.setOnClickListener {
                val intentBind = intentFor<BindService>("request_content" to
et_request.text.toString())
                //以绑定方式启动服务
                val bindFlag = bindService(intentBind, mFirstConn,
Context.BIND_AUTO_CREATE)
                Log.d(TAG, "bindFlag=" + bindFlag)
                toast("服务已绑定启动")
            }
            btn_unbind.setOnClickListener {
                if (mBindService != null) {
                    //解除绑定服务
                    unbindService(mFirstConn)
                    mBindService = null
                    toast("服务已解除绑定")
                }
            }
        }

        private var mBindService: BindService? = null
        private val mFirstConn = object : ServiceConnection {
```

```
            //获取服务对象时的操作
            override fun onServiceConnected(name: ComponentName, service: IBinder) {
                //如果服务运行于另一个进程，就不能直接强制转换类型
                //否则会报错“java.lang.ClassCastException: android.os.BinderProxy
cannot be cast to...”
                mBindService = (service as BindService.LocalBinder).service
                Log.d(TAG, "onServiceConnected")
            }

            //无法获取服务对象时的操作
            override fun onServiceDisconnected(name: ComponentName) {
                mBindService = null
                Log.d(TAG, "onServiceDisconnected")
            }

        }

        companion object Bind {
            private val TAG = "ServiceBindActivity"
            private var tv_bind: TextView? = null
            private var mDesc = ""

            fun showText(desc: String) {
                mDesc = "$mDesc${DateUtil.nowTime} $desc\n"
                tv_bind?.text = mDesc
            }
        }
    }
```

下面是待绑定/解绑的 Kotlin 服务代码例子：

```
    class BindService : Service() {
        private val mBinder = LocalBinder()

        inner class LocalBinder : Binder() {
            val service: BindService
                get() = this@BindService
        }

        override fun onCreate() {
            ServiceBindActivity.showText("创建服务")
            super.onCreate()
        }

        override fun onBind(intent: Intent): IBinder? {
            val bundle = intent.extras
```

```
        val request_content = bundle.getString("request_content")
        ServiceBindActivity.showText("绑定服务，收到请求内容：
${request_content}")
        return mBinder
    }

    override fun onUnbind(intent: Intent): Boolean {
        ServiceBindActivity.showText("解绑服务")
        return true
    }
}
```

上述绑定方式启停服务的运行效果如图 9-27 和图 9-28 所示，其中图 9-27 所示为绑定服务之后的界面，图 9-28 所示为解绑服务之后的界面。

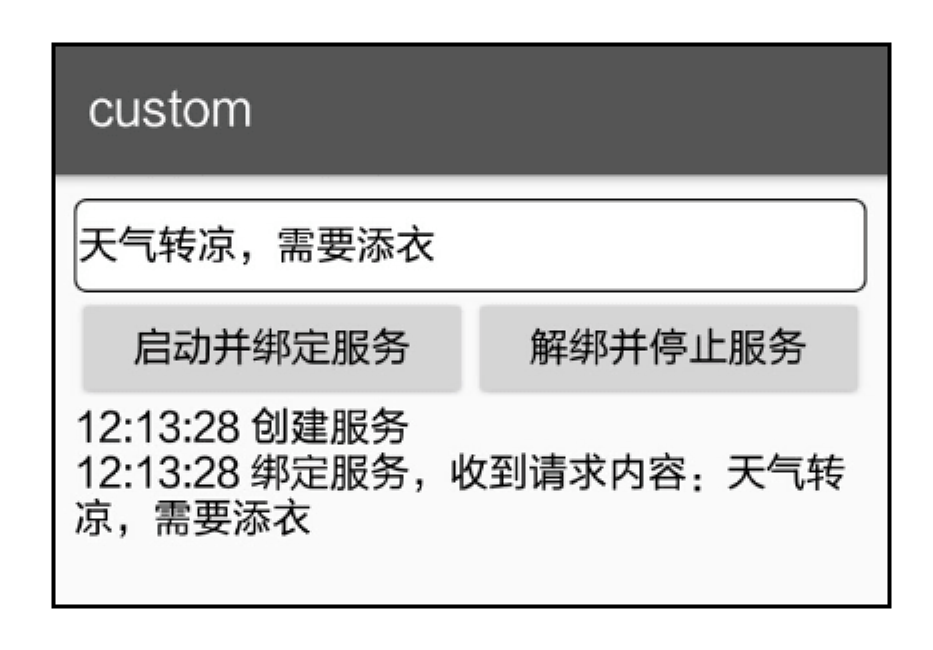


图 9-27　服务绑定的效果



图 9-28　服务解绑的效果

9.4.3　推送服务到前台

前两个小节为了观察服务的运行情况强行调用了 Activity 类的静态方法，好让页面显示服务的运行情况。可是这种做法很不安全，因为页面随时都会跳转或者干脆销毁，此时服务就失去了界面寄托。所以更好的做法是，不要让服务依附于任何页面，服务只管做好自己，而 Android 允许服务以某种形式出现在屏幕上，这个呈现服务的形式便是通知栏。

是否让服务显示到通知栏上面，需要在服务内部执行下面的两个前台方法，说明如下。

- startForeground：把当前服务切换到前台运行。第一个参数表示通知的编号，第二个参数表示 Notification 对象，这意味着切换到前台就是展示到通知栏。
- stopForeground：停止前台运行。参数为 true 表示清除通知，为 false 表示不清除。

服务在前台运行的一个常见应用是音乐播放器，即使用户离开了播放器页面，手机也能在后台继续播放音乐，同时还能在通知栏查看播放进度以及控制播放与暂停操作。下面是一个音乐播放服务的 Kotlin 代码例子：

```
class MusicService : Service() {
    inner class LocalBinder : Binder() {
        val service: MusicService
```

```
            get() = this@MusicService
    }

    private val mBinder = LocalBinder()
    override fun onBind(intent: Intent): IBinder? = mBinder

    private var mSong: String = ""
    private var PAUSE_EVENT = ""
    private var isPlay = true
    private var mBaseTime: Long = 0
    private var mPauseTime: Long = 0
    private var mProgress = 0
    private val handler = Handler()
    private val playTask = object : Runnable {
        override fun run() {
            if (isPlay) {
                if (mProgress < 100) {
                    mProgress += 2
                } else {
                    mProgress = 0
                }
                handler.postDelayed(this, 1000)
            }
            val notify = getNotify(this@MusicService, PAUSE_EVENT, mSong, isPlay,
mProgress, mBaseTime)
            //持续刷新通知栏上的播放进度
            startForeground(2, notify)
        }
    }

    private fun getNotify(ctx: Context, event: String, song: String, isPlay:
Boolean, progress: Int, time: Long): Notification {
        val pIntent = Intent(event)
        val nIntent = PendingIntent.getBroadcast(ctx,
                R.string.app_name, pIntent, PendingIntent.FLAG_UPDATE_CURRENT)
        val notify_music = RemoteViews(ctx.packageName, R.layout.notify_music)
        if (isPlay) {
            notify_music.setTextViewText(R.id.btn_play, "暂停")
            notify_music.setTextViewText(R.id.tv_play, "${song}正在播放")
            notify_music.setChronometer(R.id.chr_play, time, "%s", true)
        } else {
            notify_music.setTextViewText(R.id.btn_play, "继续")
            notify_music.setTextViewText(R.id.tv_play, "${song}暂停播放")
            notify_music.setChronometer(R.id.chr_play, time, "%s", false)
        }
```

```
        notify_music.setProgressBar(R.id.pb_play, 100, progress, false)
        notify_music.setOnClickPendingIntent(R.id.btn_play, nIntent)
        val intent = ctx.intentFor<MainActivity>()
        val cIntent = PendingIntent.getActivity(ctx,
                R.string.app_name, intent, PendingIntent.FLAG_UPDATE_CURRENT)
        val builder = Notification.Builder(ctx)
        return builder.setContentIntent(cIntent)
                .setContent(notify_music)
                .setTicker(song)
                .setSmallIcon(R.drawable.tt_s).build()
    }

    override fun onStartCommand(intent: Intent, flags: Int, startid: Int): Int {
        mBaseTime = SystemClock.elapsedRealtime()
        isPlay = intent.getBooleanExtra("is_play", true)
        mSong = intent.getStringExtra("song")
        handler.postDelayed(playTask, 200)
        return Service.START_STICKY
    }

    override fun onCreate() {
        PAUSE_EVENT = resources.getString(R.string.pause_event)
        pauseReceiver = PauseReceiver()
        registerReceiver(pauseReceiver, IntentFilter(PAUSE_EVENT))
        super.onCreate()
    }

    override fun onDestroy() {
        unregisterReceiver(pauseReceiver)
        super.onDestroy()
    }

    private var pauseReceiver: PauseReceiver? = null
    //定义一个处理播放/暂停事件的广播接收器内部类
    inner class PauseReceiver : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent?) {
            if (intent != null) {
                isPlay = !isPlay
                if (isPlay) {
                    handler.postDelayed(playTask, 200)
                    if (mPauseTime > 0) {
                        val gap = SystemClock.elapsedRealtime() - mPauseTime
                        mBaseTime += gap
                    }
                } else {
```

```
                        mPauseTime = SystemClock.elapsedRealtime()
                    }
                }
            }
        }
    }
```

上述 Kotlin 代码的与众不同之处在于点击“播放/暂停”按钮的处理，此时触发的延迟意图对象由 getBroadcast 方法获得，原因是 getActivity 获得的对象只会跳到某个页面，要想让触发的事件作用于服务内部，只能通过广播的方式。

下面是启动和停止音乐播放服务的 Kotlin 页面代码示例：

```
class NotifyServiceActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_notify_service)
        var bPlay = false
        btn_send_service.setOnClickListener {
            bPlay = !bPlay
            //声明携带两个输入参数的意图对象
            val intent = intentFor<MusicService>("is_play" to bPlay,
                    "song" to et_song.text.toString())
            if (bPlay) {
                startService(intent)
                toast("歌曲${et_song.text}已在通知栏开始播放")
                btn_send_service.text = "停止播放音乐"
            } else {
                stopService(intent)
                toast("歌曲${et_song.text}已从通知栏清除")
                btn_send_service.text = "开始播放音乐"
            }
        }
    }
}
```

音乐播放服务的前台运行效果如图 9-29 和图 9-30 所示，其中图 9-29 所示为正在播放中的通知栏界面，图 9-30 所示为暂停播放时的通知栏界面。

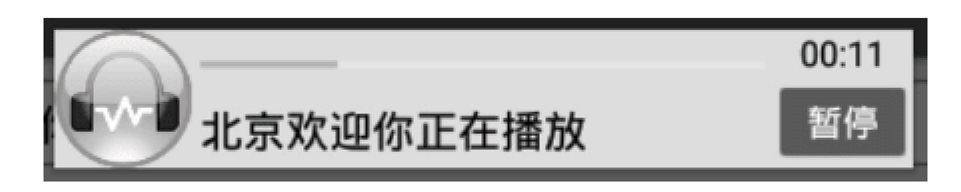



图 9-29 正在播放中的通知栏界面

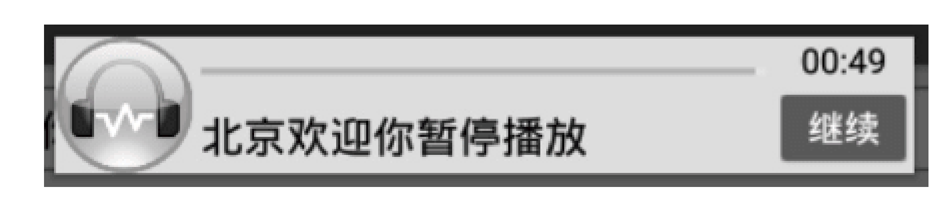


图 9-30 暂停播放时的通知栏界面

9.5　实战项目：电商 App 的生鲜团购

现在的电子商务 App 几乎是无所不卖，从服装到图书，从家电到家具，从数码到日用品，越来越多的商品在网上销售。然而，有一大品类的商品领域迟迟未能诞生独角兽公司，这便是生鲜电商。即使巨头打造了盒马鲜生、京东到家等业态，也没有实现市场的成熟化，这是为什么呢？由于生鲜有着与其他商品不同的特点，比如生鲜的保质期很短，造成商家无法长期储存；又比如生鲜的价格受时令影响，容易起伏波动；再比如生鲜的销售是讲究规模的，大量批发才有利可图等。因此这些特点决定了生鲜产品无法像其他商品那样采取常规方式销售，而必须采用新方式进行销售，新的销售方式建议具备“预订+团购+统一发货”的特征，从而既摊薄顾客的成本又降低商家的风险。本章结尾通过“电商 App 的生鲜团购”这个实战项目详细分析如何利用 App 开发实现生鲜团购的相关功能。

9.5.1　需求描述

需求分析不是一个轻松的活儿，因为首先要表达清楚没有遗漏功能，其次要循循善诱，让技术人员明白这是什么事儿，再次还得有条理、成体系，方便做出原型来。既然如此，不妨先看几张界面效果图，有个直观印象更容易产生联想。生鲜团购的页面一进来就展示几个热卖食品，先声夺人，紧紧抓住吃货们的眼球，如图 9-31 所示，这是生鲜食品的列表页面，鲜艳的大闸蟹和小龙虾美食令人垂涎欲滴；把列表页往上拉，继续展示剩余的虾类以及名贵鱼类的饕餮大餐，如图 9-32 所示。



图 9-31　生鲜食品的列表页面效果



图 9-32　生鲜列表上拉之后的界面

看到一款令人心仪已久的阳澄湖大闸蟹，点击它跳到大闸蟹的详情页面，可见当前已有 500 人加入团购，如图 9-33 所示。还有什么能阻挡吃货们的大快朵颐呢？赶紧点击“立即参加团购”按钮，页面提示成功参团（已团购人数加一），如图 9-34 所示。



图 9-33 生鲜食品的详情页面



图 9-34 参加团购后的详情页

可惜未达到团购的目标人数，还需耐心等待其他人的加入。但是用户又不可能一直停留在这个团购页面，因此应当由 App 自己想办法实时获取参团人数，并将最新的参团人数推送到通知栏，方便用户时不时地关心一下，通知栏的推送效果如图 9-35 所示。随着时间的流逝，终于在某个时刻参团人数达到 1000 人，此时系统赶忙震动手机提醒用户，如图 9-36 所示。



图 9-35 团购人数实时获取中



图 9-36 团购人数已达组团要求

正常组团成功之后，还要用户支付定金，如此才算完成生鲜团购的预订流程。不过这里仅仅是练手而已，所以能够实现上述的界面效果即可。

9.5.2 开始热身：震动器 Vibrator

9.5.1 小节的需求描述提到，团购人数达到目标数量时，手机要立即震动，提示用户组团成功。使用震动器要在 AndroidManifest.xml 中加入如下权限：

```
<!-- 震动 -->
<uses-permission android:name="android.permission.VIBRATE" />
```

让手机震动的功能用到了震动器 Vibrator 类，而震动器对象从系统服务 VIBRATOR_SERVICE 获得，实现该功能的代码很简单，即便用 Java 书写也只有以下两行代码：

```
        Vibrator vibrator = (Vibrator) getSystemService
(Context.VIBRATOR_SERVICE);
        vibrator.vibrate(3000);
```

两行代码看起来真没什么好简化的了，因为转换成 Kotlin 也要下面的两行代码：

```
        //常规做法：从系统服务中获取震动器对象
        val vibrator = getSystemService(Context.VIBRATOR_SERVICE) as Vibrator
        vibrator.vibrate(3000)
```

虽然获取震动器的代码并不多，但是这真的很难记忆，首先开发者要调用 getSystemService 方法，接着绞尽脑汁才能想起该服务的名称是 VIBRATOR_SERVICE，最后将类型强制转换为 Vibrator。其中，即有大写字母又有小写字母还有大小写混合，对于英文不溜的朋友来说，这简直是个灾难。如果只要一个朗朗上口的单词就能代表震动器，那势必会为开发者省去背诵专业英语单词的麻烦。然而两行代码还能怎么优化？倘若改造成工具类获取震动器对象，也不见得一定省事。

不过 Kotlin 可不会善罢甘休，相反是迎难而上，因为它坐拥扩展函数这个法宝，之前我们多次见识了扩展函数的威力，比如提示窗的 toast、提醒对话框的 alert 等。当然，获取震动器对象也能按照扩展函数来改造，比如给 Context 添加一个扩展函数 getVibrator，该扩展函数的 Kotlin 代码示例如下：

```
    //获取震动器
    fun Context.getVibrator() : Vibrator {
        return getSystemService(Context.VIBRATOR_SERVICE) as Vibrator
    }
```

接着回到 Activity 页面代码，实现震动功能只需下面的一行代码：

```
        //利用扩展函数获得震动器对象
        getVibrator().vibrate(3000)
```

以上代码固然简化了，却仍然不是最简单的写法，看看 getVibrator()方法，前面有 get 后面有括号，都是碍手碍脚的家伙。可去掉括号就不是函数了，而变成了属性，难不成 Kotlin 什么时候多了个扩展属性的用法？其实 Kotlin 还真的可以实现扩展属性的功能，关键是要利用扩展函数进行移花接木。首先要在 kt 文件中声明一个 Context 类的新属性，然后定义该属性的 get 方法（get 方法为扩展函数）。如此一来，外部访问该扩展属性时，编译器会自动调用该属性的 get 方法，从而通过扩展函数间接实现扩展属性。

接下来依旧以震动器为例，看看如何使用 Kotlin 代码声明扩展属性 vibrator：

```
    //获取震动器
    //利用扩展函数实现扩展属性，在 Activity 代码中即可直接使用 vibrator
    val Context.vibrator : Vibrator
        get() = getSystemService(Context.VIBRATOR_SERVICE) as Vibrator
```

现在回到 Activity 代码，只要通过 vibrator 就能访问震动器的方法，如下所示：

```
        //利用扩展函数实现扩展属性，直接使用 vibrator 即可指代震动器对象
        vibrator.vibrate(3000)
```

当然，要想正常访问自定义的扩展函数和扩展属性，需要在 Activity 代码头部加入以下的导入语句：

```
    import com.example.custom.util.vibrator
```

除了震动器之外，其他从系统服务获得对象的管理器也能照此办理，譬如本章“9.3 自定义通知栏”提到的通知管理器 NotificationManager，按照之前的调用方式是下面的 Kotlin 代码：

```
           val notifyMgr = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
           notifyMgr.notify(R.string.app_name, notify)
```

显然，通知管理器对象的获取代码更冗长，接下来将其改造为扩展属性的方式，则相应的 Context 扩展代码如下所示：

```
    //获取通知管理器
    //试试在 Activity 代码中调用“notifier.notify(R.string.app_name, notify)”
    val Context.notifier: NotificationManager
       get() = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
```

原来获取通知管理器的两行代码便缩减为下面的一行 Kotlin 代码了：

```
           notifier.notify(R.string.app_name, notify)
```

举一反三，来自系统服务的其余管理器统统运用扩展属性，能够更加方便将来的开发工作。下面是几个常用管理器通过扩展属性改写后的 Kotlin 实现代码：

```
    //获取下载管理器
    val Context.downloader: DownloadManager
       get() = getSystemService(Context.DOWNLOAD_SERVICE) as DownloadManager
    //获取定位管理器
    val Context.locator: LocationManager
       get() = getSystemService(Context.LOCATION_SERVICE) as LocationManager
    //获取连接管理器
    val Context.connector: ConnectivityManager
       get() = getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager
    //获取电话管理器
    val Context.telephone: TelephonyManager
       get() = getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager
    //获取无线管理器
    val Context.wifi: WifiManager
       get() = getSystemService(Context.WIFI_SERVICE) as WifiManager
    //获取闹钟管理器
    val Context.alarm: AlarmManager
       get() = getSystemService(Context.ALARM_SERVICE) as AlarmManager
```

```
//获取音频管理器
val Context.audio: AudioManager
    get() = getSystemService(Context.AUDIO_SERVICE) as AudioManager
```

9.5.3　控件设计

正如生鲜团购是个崭新的事物，这里的实战项目用到了多个重新定义的控件，其实也不必标新立异，只要把本章前面介绍过的自定义控件直接拿过来，复习复习它们的实现步骤及其用法即可。这些相关的自定义控件概述如下。

- 不滚动列表视图 NoScrollListView：一个页面嵌入多个列表，每个列表都要完全展示，只能使用自定义的 NoScrollListView。
- 圆角布局 RoundLayout：每个列表内的商品都属于同一种类，最好能够与周围区域通过界线分隔开，暂时采用 RoundLayout。
- 进度条 ProgressBar：商品详情页面可展示团购进度，通知栏上也可展示团购进度，二者都用到了进度条 ProgressBar。
- 任务 Runnable：要想通过动画形式渲染团购人数的当前进度，可利用任务 Runnable 持续刷新进度条。
- 通知推送 Notification：把团购信息推送到通知栏，需要采用通知推送 Notification。
- 远程视图 RemoteViews：自定义的消息通知必须经由 RemoteViews 实现该通知的控件布局。

除了上面的几个自定义控件外，按照效果图还需要一个团购服务在后台运行，模拟团购信息的实时刷新效果。另外，本项目还使用了几个系统服务，结合 App 自定义的服务说明如下。

- 服务 Service：后台运行的团购服务不但要模拟团购信息的实时交互，而且要借助通知栏推送到前台展示给用户观看。
- 通知管理器 NotificationManager：通知管理器的对象实例从系统服务 NOTIFICATION_SERVICE 中获得，它用于管理通知的推送与回收。
- 震动器 Vibrator：震动器的对象实例从系统服务 VIBRATOR_SERVICE 中获得，它用于控制手机震动的时长。

瞅一瞅着实不简单，别看生鲜团购貌似只有两个页面，实际上用到的开发技术却不少。

9.5.4　关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成生鲜团购项目，下面列举几个重要功能的 Kotlin 代码片段。

1. 关于初始化生鲜商品的列表信息

因为一个页面可能展示多个生鲜列表，所以每个生鲜列表需要使用“9.1.2 测量尺寸”小节提到的不滚动列表视图 NoScrollListView 来展示。另外，构造生鲜商品的列表数据可采用 Kotlin 增强了的可变队列 MutableList，且队列中的每项记录可以采取命名参数来声明。

下面是初始化生鲜列表的 Kotlin 代码示例：

```
    private fun initCrabList() {
        val freshList = mutableListOf<FreshInfo>(
                FreshInfo(name = "阳澄湖大闸蟹",
                        desc = "产自阳澄湖的天然大闸蟹，口味一流，认准阳澄湖。",
                        imageId = R.drawable.dazhaxie,
                        price = 999, peopleCount = 500 ),
                FreshInfo(name = "平潭红鲟",
                        desc = "膏红肉肥的锯缘青蟹，滋补强身，平潭特产。",
                        imageId = R.drawable.hongxun,
                        price = 666, peopleCount = 500 ),
                FreshInfo(name = "阿拉斯加帝王蟹",
                        desc = "来自大自然的馈赠，阿拉斯加当地深海捕捞。",
                        imageId = R.drawable.diwangxie,
                        price = 888, peopleCount = 500 ) )
        val adapter = FreshAdapter(this, freshList)
        nslv_crab.adapter = adapter
        nslv_crab.onItemClickListener = adapter
    }
```

2. 关于携带生鲜信息跳转至详情页面

在生鲜列表页面，点击某个生鲜商品会跳转到该商品的详情页面，在跳转的同时要携带生鲜信息的参数。围绕这个参数的携带过程，Kotlin 代码应当进行以下修改：

（1）定义一个生鲜信息的 Parcelable 类，并使用注解“@Parcelize”修饰该类。下面是 Parcelable 类的 Kotlin 定义代码（若函数体没有代码，则可省略函数体的大括号）：

```
    @Parcelize
    data class FreshInfo(var name: String="", var desc: String="", var imageId:
Int=0,
                var price: Int=0, var peopleCount: Int=0, var isJoin:
Boolean=false) : Parcelable
```

（2）在列表适配器中执行页面跳转动作，可利用 Anko 库的 startActivity 函数完成跳转操作，下面是 Kotlin 的页面跳转代码例子：

```
        context.startActivity<FreshDetailActivity>("fresh" to fresh)
```

（3）详情页面调用 Intent 对象的 getParcelableExtra 方法读取列表页传来的生鲜数据，下面是 Kotlin 读取请求参数的代码例子：

```
        var freshInfo: FreshInfo = intent.getParcelableExtra("fresh")
```

3. 关于提前终止进度条动画的播放

用户在生鲜详情页面点击“立即参加团购”按钮后，界面上的进度条动画应当立即停止，并且进度条文字上的团购数量同时加一。此时肯定要调用处理器 Handler 对象的 removeCallbacks 方

法来回收 Runnable 任务，另外要延迟一定时间再去设置进度条上的团购文本，避免任务回收的异步处理造成代码的执行顺序产生混乱。

下面是停止进度动画（即回收任务对象）的 Kotlin 代码：

```
    handler.removeCallbacks(animation)
    handler.postDelayed({
        tpb_count.progress = (count+1)*100/TOTAL
        tpb_count.progressText = "当前已有${count+1}人加入团购"
        tpb_count.invalidate()  //进度条属性发生变化，调用 invalidate 方法立即刷新当前进度
    }, 100)
```

4. 关于携带团购信息启动团购服务

用户点击“立即参加团购”按钮，此时后台自动启动团购服务，并模拟与服务器的团购信息交互，为此也要把团购信息传递给团购服务。这里依旧利用 Anko 库的 startService 函数完成服务启动操作，具体的 Kotlin 启动代码如下所示：

```
        startService<GroupService>("fresh" to freshInfo)
```

5. 关于服务销毁时回收通知推送

如果用户退出电商 App，那么为了减少系统资源消耗，可在团购服务销毁时回收通知栏上的团购消息。回收服务通知的 Kotlin 代码示例如下：

```
    override fun onDestroy() {
        super.onDestroy()
        if (notify != null) {
            stopForeground(true)  //停止前台运行的同时清除通知
        }
    }
```

9.6　小　结

本章主要介绍了 Kotlin 如何完成几种自定义控件的实现过程，包括自定义普通视图的三个步骤（构造对象、测量尺寸、绘制部件）、简单动画的自定义实现（任务 Runnable、进度条 ProgressBar 以及进度条动画的实现）、通知推送的展现形式（常规通知、大视图通知、三种特殊通知、自定义通知、折叠式通知）以及 Service 服务组件的启停方式（普通启停、绑定与解绑、推送到前台）。最后设计了一个实战项目“电商 App 的生鲜团购”，在该项目的 Kotlin 编码中采用了前面介绍的部分自定义控件以及 Service 服务的启停和推送，另外还介绍了 Kotlin 对震动器用法的改进编码。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会使用 Kotlin 完成自定义视图的实现过程，除了构造对象、测量尺寸、绘制部件的三大步骤之外，重点掌握 Kotlin 的主构造函数和注解“@JvmOverloads”在自定义视图中的运用。

（2）学会使用 Kotlin 实现简单的自定义动画，重点掌握 Kotlin 对任务对象 Runnable 的 4 种定义方式，并利用 Runnable 实现简单的进度条动画。

（3）学会使用 Kotlin 展示不同形式的消息通知，包括常规通知、大视图通知、三种特殊通知（进度通知、浮动通知、锁屏通知）、自定义通知、折叠式通知等。

（4）学会使用 Kotlin 运用 Service 服务的三种启停方式，包括普通方式的启停、绑定方式的启停、推送到通知栏与从通知栏回收，重点掌握 Kotlin 如何携带请求参数启动服务。

（5）学会使用 Kotlin 的扩展函数特性实现扩展属性，并运用扩展属性简化各种系统管理器的实例获取写法。

第 10 章

Kotlin 实现网络通信

本章将介绍 Kotlin 实现网络通信功能的相关技术，包括多线程技术的运用、HTTP 接口的访问操作、文件下载的处理方式，另外还将介绍安卓四大组件之一内容提供器 ContentProvider 的常见用法。最后结合本章所学的知识演示一个实战项目“电商 App 的自动升级”的设计与实现。

10.1　多线程技术

手机应用与传统软件有一个很大的区别，就是 App 很讲究画面的流畅度，毕竟手机屏幕只比豆腐块略大一些，在这有限的方寸之间，如果发生卡顿乃至卡死的情况，对于用户来说简直是不可忍受的。因此，为了保证 App 画面的流畅，同时也要兼顾事务的正常处理，引进多线程技术便是不可或缺的了。但是 Android 又规定分线程不能直接操作界面控件，于是围绕着如何启动分线程、如何维持线程间的信息交互，从而衍生出处理器消息机制、进度对话框提示、异步任务处理等技术。接下来本节将对这些多线程相关技术进行深入的分析和介绍。

10.1.1　大线程 Thread 与消息传递

App 开发时常会遇到一些耗时的业务场景，比如后台批量处理数据、访问后端服务器接口等，此时为了保证界面交互的及时响应，必须通过分线程单独运行这些耗时任务。简单的线程可使用 Thread 类来启动，无论是 Java 还是 Kotlin 都一样，该方式首先要声明一个自定义线程类，对应的 Java 代码如下所示：

```
    private class PlayThread extends Thread {
        @Override
        public void run() {
```

```
            //此处省略具体的线程内部代码
        }
    }
```

自定义线程类的 Kotlin 代码与 Java 大同小异，具体如下：

```
    private inner class PlayThread : Thread() {
        override fun run() {
            //此处省略具体的线程内部代码
        }
    }
```

线程类声明完毕，接着要启动线程处理任务，在 Java 中调用一行代码“new PlayThread().start();”即可，至于 Kotlin 则更简单，只要调用“PlayThread().start()”就行。如此看来，Java 的线程处理代码跟 Kotlin 差不了多少，没发觉 Kotlin 比 Java 有什么优势。倘使这样，真是小瞧了 Kotlin，它身怀多项绝技，单单是匿名实例这招，之前在“9.2.1 任务 Runnable”小节便领教过了，同样线程 Thread 也能运用匿名实例方式化繁为简。注意到自定义线程类均需由 Thread 派生而来，然后必须且仅需重写 run 函数，所以像类继承、函数重载这些代码都是走过场，完全没必要每次都依样画葫芦，编译器真正关心的只是 run 函数内部的具体代码。

于是，借助匿名实例的手段，Kotlin 的线程执行代码可以简写成下面这般：

```
        Thread {
            //此处省略具体的线程内部代码
        }.start()
```

以上的 Kotlin 代码段看似无理，实则有规，不但指明这是个线程，而且命令启动该线程，可谓是简洁明了。

线程代码在运行过程中，通常还要根据实际情况来更新界面，以达到动态刷新的效果。可是 Android 规定了只有主线程才能操作界面控件，分线程是无法直接使用控件对象的，只能通过 Android 提供的处理器 Handler 才能间接操纵控件。这意味着，要想让分线程持续刷新界面，仍需完成传统 Android 开发的下面几项工作：

（1）声明一个自定义的处理器类 Handler，并重写该类的 handleMessage 函数，根据不同的消息类型进行相应的控件操作。

（2）线程内部针对各种运行状况，调用处理器对象的 sendEmptyMessage 或者 sendMessage 方法发送事先约定好的消息类型。

举个具体的业务例子，现在有一个新闻版块，每隔两秒在界面上滚动播报新闻，其中便联合运用了线程和处理器，先由线程根据实际情况发出消息指令，再由处理器按照消息指令轮播新闻。详细的 Kotlin 页面代码示例如下：

```
class MessageActivity : AppCompatActivity() {
    private var bPlay = false
    private val BEGIN = 0 //开始播放新闻
    private val SCROLL = 1 //持续滚动新闻
    private val END = 2 //结束播放新闻
```

```
        private val news = arrayOf("北斗三号卫星发射成功，定位精度媲美 GPS", "美国赌城拉斯维加斯发生重大枪击事件", "日本在越南承建的跨海大桥未建完已下沉", "南水北调功在当代，数亿人喝上长江水", "马克龙呼吁重建可与中国匹敌的强大欧洲")

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_message)
            //指定文本视图内部文本的对齐方式为靠左且靠右对齐
            tv_message.gravity = Gravity.LEFT or Gravity.BOTTOM
            //指定文本视图的显示行数为 8 行
            tv_message.setLines(8)
            //指定文本视图的最大行数为 8 行
            tv_message.maxLines = 8
            //指定文本视图内部文本的移动方式为滚动
            tv_message.movementMethod = ScrollingMovementMethod()
            btn_start_message.setOnClickListener {
                if (!bPlay) {
                    bPlay = true
                    //线程第一种写法的调用方式通过具体的线程类进行构造
                    //注意每个线程实例只能启动一次，不能重复启动
                    //若要多次执行该线程的任务，则需每次都构造新的线程实例
                    //PlayThread().start()
                    //线程的第二种写法，采用匿名实例的形式。第二种写法无须显式构造
                    Thread {
                        //发送“开始播放新闻”的消息类型
                        handler.sendEmptyMessage(BEGIN)
                        while (bPlay) {
                            //休眠两秒，模拟获取突发新闻的网络延迟
                            Thread.sleep(2000)
                            //调用 Message 的 obtain 方法，获得一个消息实例
                            val message = Message.obtain()
                            message.what = SCROLL
                            message.obj = news[(Math.random() * 30 % 5).toInt()]
                            //发送“持续滚动新闻”的消息类型
                            handler.sendMessage(message)
                        }
                        bPlay = true
                        Thread.sleep(2000)
                        //发送“结束播放新闻”的消息类型
                        handler.sendEmptyMessage(END)
                        bPlay = false
                    }.start()
                }
            }
            btn_stop_message.setOnClickListener { bPlay = false }
```

```
    }

    //线程的第一种写法，继承 Thread 类并重载 run 方法
//    private inner class PlayThread : Thread() {
//        override fun run() {
//            handler.sendEmptyMessage(BEGIN)
//            while (bPlay) {
//                Thread.sleep(2000)
//                val message = Message.obtain()
//                message.what = SCROLL
//                message.obj = news[(Math.random() * 30 % 5).toInt()]
//                handler.sendMessage(message)
//            }
//            bPlay = true
//            Thread.sleep(2000)
//            handler.sendEmptyMessage(END)
//            bPlay = false
//        }
//    }

    //自定义的处理器类，区分三种消息类型，给 tv_message 显示不同的文本内容
    private val handler = object : Handler() {
        override fun handleMessage(msg: Message) {
            val desc = tv_message.text.toString()
            tv_message.text = when (msg.what) {
                BEGIN -> "$desc\n${DateUtil.nowTime} 下面开始播放新闻"
                SCROLL -> "$desc\n${DateUtil.nowTime} ${msg.obj}"
                else -> "$desc\n${DateUtil.nowTime} 新闻播放结束，谢谢观看"
            }
        }
    }
}
```

上述滚动播报新闻的运行效果如图 10-1 和图 10-2 所示，其中图 10-1 展示正在播放新闻的界面，此时分线程每隔两秒添加一条新闻；图 10-2 展示新闻播放结束时的界面，此时主线程收到分线程的 END 消息，于是提示用户“新闻播放结束，谢谢观看”。

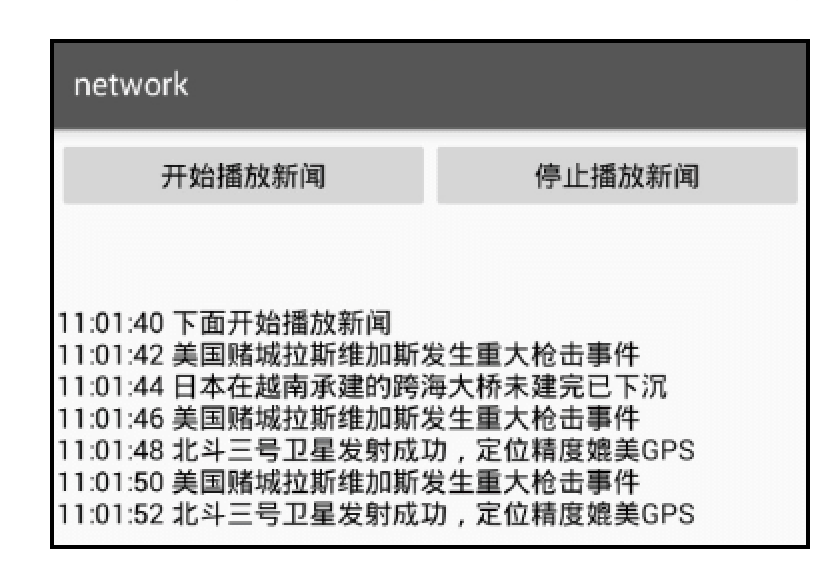


图 10-1　正在播放新闻的界面

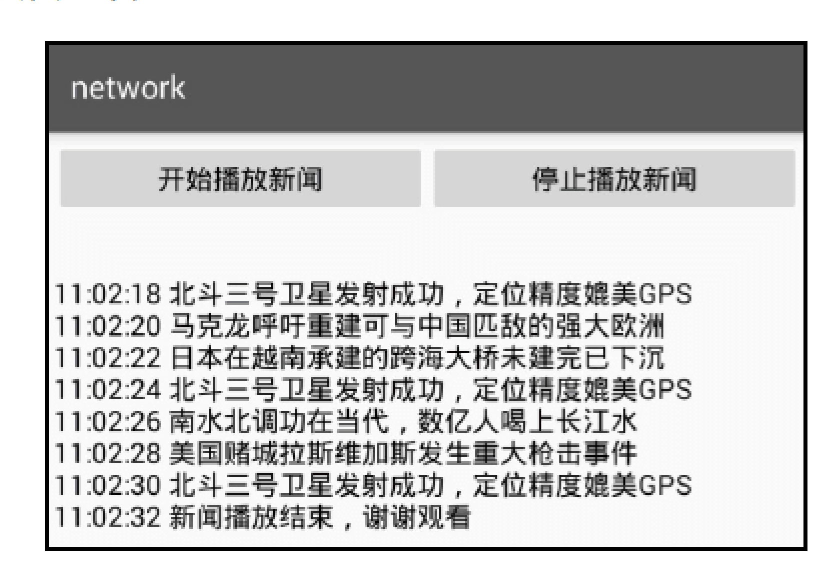


图 10-2　新闻播放结束的界面

10.1.2　进度对话框 ProgressDialog

手机 App 访问接口、加载网页之类的请求服务端行为基本上属于耗时操作，慢的时候要过好几秒才能加载完毕。在此期间，为了减少用户的等待焦灼感，界面需要展示正在加载的动画，一方面避免产生 App 卡死的错觉，另一方面提示用户耐心等待。这时候就用到了进度对话框，通过让 App 在加载开始前弹出进度框，加载结束后关闭进度框，从而改善加载交互的用户体验。

进度对话框分两种，一种是水平进度对话框，另一种是圆圈进度对话框。虽然在第 9 章的“9.2.2 进度条 ProgressBar”提到了进度条控件，但是 ProgressBar 只是一个单独的控件，必须在页面上占好位置才会显示。而 ProgressDialog 把 ProgressBar 封装到了对话框里面，有需要提示的时候才弹窗，没需要了就关闭窗口，所以进度对话框比进度条更适用于等待行为。下面对这两种进度对话框分别进行介绍。

1. 水平进度对话框

水平进度对话框允许实时刷新当前进度，方便用户知晓已处理的进展百分比。它主要包含消息标题、消息内容、对话框样式（水平还是圆圈）、当前进度这 4 种元素，若使用 Java 代码实现该对话框，则是很常规的编码风格，具体的 Java 代码举例如下：

```
ProgressDialog dialog = new ProgressDialog(this);
dialog.setTitle("请稍候");
dialog.setMessage("正在努力加载页面");
dialog.setMax(100);
dialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
dialog.show();
```

水平进度对话框的 Java 编码看起来中规中矩，可是仍然显得拖泥带水，很简单的功能也花费了 6 行 Java 代码。倘若使用 Kotlin 书写，则借助于 Anko 库只需下面两行代码：

```
val dialog = progressDialog("正在努力加载页面", "请稍候")
dialog.show()
```

瞧瞧，水平进度对话框的实现代码顿时变得清爽了许多，其界面效果与 Java 是完全一样的。当然，因为用到了 Anko 库的扩展函数，所以务必在代码头部加入一行导入语句：

```
import org.jetbrains.anko.progressDialog
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

在水平进度对话框弹出之后，若想更新水平条的进度值，则可调用以下一行 Kotlin 代码设置当前进度：

```
dialog.progress = 10 //进度值（取值为 0～100）
```

当进度值达到 100 时，意味着耗时任务处理完成，此时即可调用对话框对象的 dismiss 方法来关闭对话框。

下面展示水平进度对话框的进度变化效果，具体如图 10-3 和图 10-4 所示，其中图 10-3 表示当前处理进度为 20%，图 10-4 表示当前处理进度已经到了 70%。



图 10-3 水平进度对话框的进度为 20%

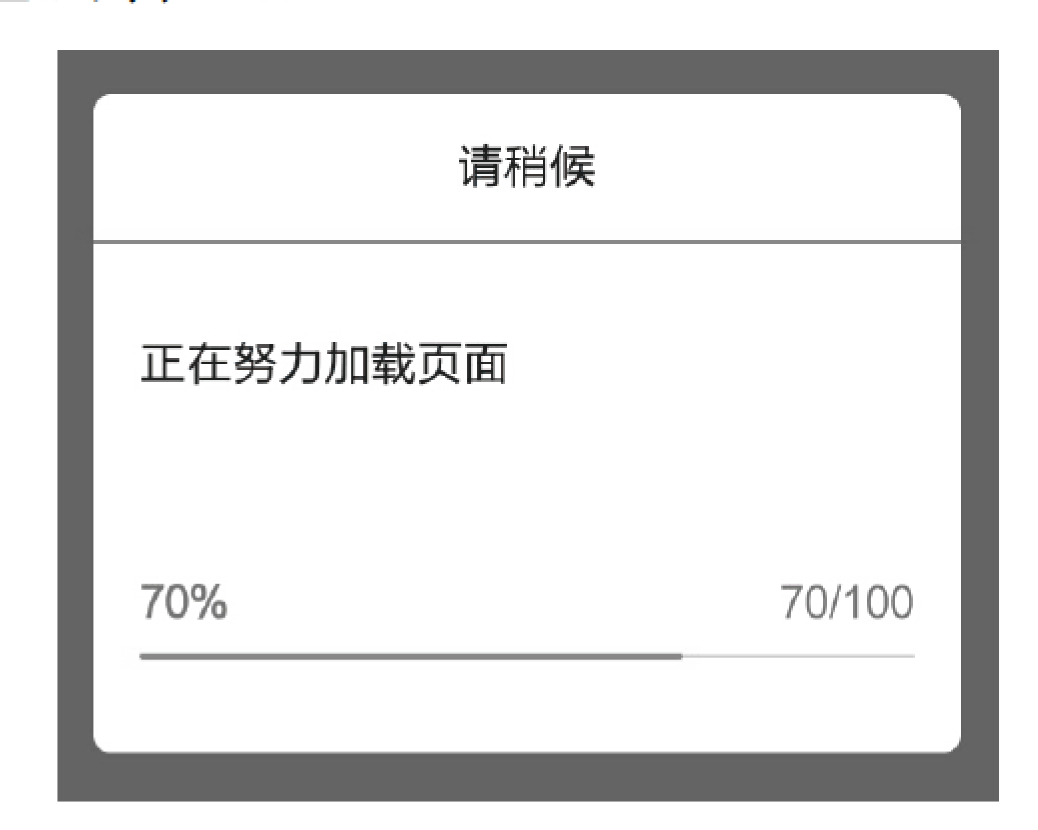


图 10-4 水平进度对话框的进度为 70%

2. 圆圈进度对话框

圆圈进度对话框仅仅展示转圈的动画效果，它不支持实时刷新处理进度，自然在编码上比水平对话框会简化一些。可是用 Java 编码显示圆圈进度对话框依旧需要下列 5 行代码：

```
ProgressDialog dialog = new ProgressDialog(this);
dialog.setTitle("请稍候");
dialog.setMessage("正在努力加载页面");
dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
dialog.show();
```

要是用 Kotlin 实现该对话框，有了水平进度对话框的的先例，不出意料只需以下两行 Kotlin 代码就行了：

```
val dialog = indeterminateProgressDialog("正在努力加载页面", "请稍候")
dialog.show()
```

注意到上面的 Kotlin 函数采取了前缀 indeterminate，该单词的意思是“模糊的、不定的”，表示这种对话框的处理进度是不确定的，不像水平进度对话框可以明确指定当前进度，据此开发者能够将 progressDialog 与 indeterminateProgressDialog 两个函数区分开。由于该函数同样来自于 Anko 库，因此不要忘了在用到的代码文件头部加入下面这行语句：

```
import org.jetbrains.anko.indeterminateProgressDialog
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

由 Kotlin 代码实现的圆圈进度对话框的转圈效果等同于 Java 代码实现的效果，具体的转圈对话框界面如图 10-5 所示。



图 10-5　圆圈进度对话框的显示效果

10.1.3　异步任务 doAsync 和 doAsyncResult

通过线程加上处理器固然可以实现滚动播放的功能，可是想必读者也看到了，这种交互方式依旧很突兀，还有好几个难以克服的缺点：

（1）自定义的处理器仍然存在类继承和函数重载的冗余写法。

（2）每次操作界面都得经过发送消息、接收消息两道工序，烦琐且拖沓。

（3）线程对象和处理器对象均需在指定的 Activity 代码中声明，无法在别处重用。

鉴于此，Android 早已提供了异步任务 AsyncTask 这个模板类，专门用于耗时任务的分线程处理。然而 AsyncTask 的用法着实不简单，首先它是个模板类，初学者瞅着模板就发慌；其次它区分了好几种运行状态，包括未运行、正在运行、取消运行、运行结束等，一堆概念令人头痛；再次为了各种状况都能与界面交互，又得定义事件监听器及其事件处理方法；末了还得在 Activity 代码中实现监听器的相应方法，才能正常调用定义好的 AsyncTask 类。

初步看了一下自定义 AsyncTask 要做的事情，直让人倒吸一口冷气，看起来很高深的样子，确实每个 Android 开发者刚接触 AsyncTask 时都费了不少脑细胞。为了说明 AsyncTask 是多么的与众不同，下面给出异步加载书籍任务的完整 Java 代码，温习一下那些年虐过开发者的 AsyncTask：

```
    //模板类的第一个参数表示外部调用 execute 方法的输入参数类型，第二个参数表示运行过程中与界面交互的数据类型，第三个参数表示运行结束后返回的输出参数类型
    public class ProgressAsyncTask extends AsyncTask<String, Integer, String> {
        private String mBook;
        //构造函数，初始化数据
        public ProgressAsyncTask(String title) {
            super();
            mBook = title;
        }

        //在后台运行的任务代码，注意此处不可与界面交互
        @Override
        protected String doInBackground(String... params) {
            int ratio = 0;
            for (; ratio <= 100; ratio += 5) {
                // 睡眠 200 毫秒模拟网络通信处理
```

```
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //刷新进度，该函数会触发调用 onProgressUpdate 方法
            publishProgress(ratio);
        }
        return params[0];
    }

    //在任务开始前调用，即先于 doInBackground 执行
    @Override
    protected void onPreExecute() {
        mListener.onBegin(mBook);
    }

    //刷新进度时调用，由 publishProgress 函数触发
    @Override
    protected void onProgressUpdate(Integer... values) {
        mListener.onUpdate(mBook, values[0], 0);
    }

    //在任务结束后调用，即后于 doInBackground 执行
    @Override
    protected void onPostExecute(String result) {
        mListener.onFinish(result);
    }

    //在任务取消时调用
    @Override
    protected void onCancelled(String result) {
        mListener.onCancel(result);
    }

    //声明监听器对象
    private OnProgressListener mListener;
    public void setOnProgressListener(OnProgressListener listener) {
        mListener = listener;
    }

    //定义该任务的事件监听器及其事件处理方法
    public static interface OnProgressListener {
        public abstract void onFinish(String result);
        public abstract void onCancel(String result);
```

```
            public abstract void onUpdate(String request, int progress, int
sub_progress);
            public abstract void onBegin(String request);
        }
    }
```

瞧瞧上面异步线程处理的 Java 代码，复杂的交互过程能叫初学者落荒而逃。见识过了 AsyncTask 的惊涛骇浪，不禁喟叹开发者的心灵有多么的强大。多线程任务是如此的令人望而却步，直到 Kotlin 与 Anko 的搭档出现，因为它俩在线程方面带来了革命性的思维，即编程理应面向产品，而非面向机器。对于分线程与界面之间的交互问题，它俩双剑合璧，给出了堪称完美的解决方案，所有的线程处理逻辑都被归结为两点：其一是如何标识这种牵涉界面交互的分线程，该点由关键字“doAsync”阐明；其二是如何在分线程中传递消息给主线程，该点由关键字“uiThread”界定。

有了这两个关键字，分线程的编码变得异乎寻常的简单，即使加上 Activity 的响应动作也只有以下寥寥数行 Kotlin 代码：

```
    private lateinit var dialog: ProgressDialog
    //展示在圆圈进度对话框
    private fun dialogCircle(book: String) {
        dialog = indeterminateProgressDialog("${book}页面加载中……", "稍等")
        doAsync {
            // 睡眠 200 毫秒模拟网络通信处理
            for (ratio in 0..20) Thread.sleep(200)
            //处理完成，回到主线程在界面上显示书籍加载结果
            uiThread { finishLoad(book) }
        }
    }

    private fun finishLoad(book: String) {
        tv_async.text = "您要阅读的《$book》已经加载完毕"
        //如果进度对话框还在显示，就关闭进度对话框
        if (dialog.isShowing) dialog.dismiss()
    }
```

以上 Kotlin 代码被 doAsync 身后大括号括起来的代码段就是分线程要执行的全部代码；至于 uiThread 身后大括号括起来的代码，则为通知主线程要完成的工作。倘若在分线程运行过程中要不断刷新当前进度，也只需在待刷新的地方添加一行 uiThread 代码便成。

下面是添加了进度刷新功能的 Kotlin 代码例子：

```
    //展示在长条进度对话框
    private fun dialogBar(book: String) {
        dialog = progressDialog("${book}页面加载中……", "稍等")
        doAsync {
            for (ratio in 0..20) {
                Thread.sleep(200)
                //处理过程中，实时通知主线程当前的处理进度
```

```
                uiThread { dialog.progress = ratio*100/20 }
            }
            uiThread { finishLoad(book) }
        }
    }
```

有些时候，App 会启动多个分线程，然后在代码中对这些线程对象进行调度，从而动态控制每个线程的运行状态。此时，doAsync 的另一个兄弟 doAsyncResult 就派上用场了，顾名思义 doAsyncResult 允许返回结果，这个结果便是一个异步线程对象，通过调用线程对象的各种查询和控制方法，即可实现人为干预线程运行的功能。

下面是使用 doAsyncResult 方法的 Kotlin 示例代码：

```
    //展示在进度条 ProgressBar
    private fun progressBar(book: String) {
        //构造异步处理需要执行的代码段 longTask，返回字符串类型
        val longTask: (AnkoAsyncContext<Context>.() -> String) = {
            for (ratio in 0..20) Thread.sleep(200)
            "加载好了" //这是 longTask 处理完成的返回结果
        }
        //doAsyncResult 返回一个异步线程对象
        val future : Future<String> = doAsyncResult(null, longTask)
        for (count in 0..10) {
            if (future.isDone) {
                //isDone 是否完成，isCancelled 是否取消，get 获取处理结果
                tv_async.text = "您要阅读的《${book}》已经${future.get()}"
                pb_async.progress = 100
                break
            }
            pb_async.progress = count*100/10
            Thread.sleep(1000)
        }
    }
```

不过因为 doAsyncResult 方法需要由开发者自行判断分线程是否处理完成，造成处理简单事务时反而显得更加麻烦，所以除了少数高级场合之外，一般并不直接调用该方法。

10.2 访问 HTTP 接口

网络编程存在着多种通信协议，常见的有传输层的 TCP 和 UDP 协议，还有应用层的诸多协议，包括用于数据与文件交互的 HTTP 协议和 FTP 协议、用于邮件收发的 SMTP 协议和 POP3 协议、用于即时通信的 XMPP 协议和 MQTT 协议等。在这些众多的网络通信协议中，最常用的当数 HTTP 协议，它不但适用于客户端与服务端之间的接口调用，也适用于客户端与服务端之间的文件传输。那么 Kotlin 又是如何实现 HTTP 协议的编程运用呢？本节接下来将从 HTTP 交互的数据格式、HTTP

接口的调用方式、HTTP 图片的获取方式这几个方面详细阐述 Kotlin 给 HTTP 编程带来的脱胎换骨的变化。

10.2.1　移动数据格式 JSON

JSON 是 App 进行网络通信时最常见的数据交互格式，Android 也自带了 JSON 格式的处理工具包 org.json，该工具包主要提供 JSONObject（JSON 对象）与 JSONArray（JSON 数组）的解析处理。下面分别介绍这两种工具类的用法。

1. JSONObject

JSONObject 的常用方法说明如下。

- 构造函数：从指定字符串构造出一个 JSONObject 对象。
- getJSONObject：获取指定名称的 JSONObject 对象。
- getString：获取指定名称的字符串。
- getInt：获取指定名称的整型数。
- getDouble：获取指定名称的双精度数。
- getBoolean：获取指定名称的布尔数。
- getJSONArray：获取指定名称的 JSONArray 数组对象。
- put：添加一个 JSONObject 对象。
- toString：把当前 JSONObject 输出为一个 JSON 字符串。

2. JSONArray

JSONArray 的常用方法说明如下。

- length：获取 JSONArray 数组对象的长度。
- getJSONObject：获取 JSONArray 数组对象在指定位置处的 JSONObject 对象。
- put：往 JSONArray 数组对象中添加一个 JSONObject 对象。

使用 JSONObject 和 JSONArray 对 JSON 串进行手工解析，处理过程比较常规，完成该功能的 Kotlin 代码与 Java 代码大同小异。下面直接给出 Kotlin 解析 JSON 串的代码片段，包括如何构造 JSON 串、如何解析 JSON 串以及如何遍历 JSON 串：

```
//构造json串
private val jsonStr: String
    get() {
        val obj = JSONObject()
        obj.put("name", "地址信息")
        val array = JSONArray()
        for (i in 0..2) {
            val item = JSONObject()
            item.put("item", "第${i+1}个元素")
            array.put(item)
        }
```

```
            obj.put("list", array)
            obj.put("count", array.length())
            obj.put("desc", "这是测试串")
            return obj.toString()
        }

    //解析 json 串
    private fun parserJson(jsonStr: String?): String {
        val obj = JSONObject(jsonStr)
        var result = "name=${obj.getString("name")}\n" +
                "desc=${obj.getString("desc")}\n" +
                "count=${obj.getInt("count")}\n"
        val listArray = obj.getJSONArray("list")
        //util 表示的范围是左闭右开区间。以下语句相当于 for (i in 0..listArray.length()
- 1)
        for (i in 0 until listArray.length()) {
            val item = listArray.getJSONObject(i)
            result = "${result}\titem=${item.getString("item")}\n"
        }
        return result
    }

    //遍历 json 串
    private fun traverseJson(jsonStr: String?): String {
        var result = ""
        val obj = JSONObject(jsonStr)
        val it = obj.keys()
        while (it.hasNext()) { // 遍历 JSONObject
            var key = it.next().toString()
            result = "${result}key=$key, value=${obj.getString(key)}\n"
        }
        return result
    }
```

上述处理 JSON 串的 Kotlin 代码对应的界面效果如图 10-6～图 10-8 所示，其中图 10-6 所示为构造 JSON 串的结果界面，图 10-7 所示为解析 JSON 串的结果界面，图 10-8 所示为遍历 JSON 串的结果界面。



图 10-6　构造 json 串的结果　　图 10-7　解析 json 串的结果　　图 10-8　遍历 json 串的结果

10.2.2　JSON 串转数据类

10.2.1 小节提到 Kotlin 对 JSON 串的手工解析没有什么好办法，其实有更高层次的办法。手工解析 JSON 串实在是麻烦，费时费力还容易犯错，所以好汉不吃眼前亏，此路难走不如另寻捷径，捷径便是甩开手工解析几条街的自动解析。

既然是自动解析，首先要制定一个规则，约定 JSON 串有哪些元素，具体对应怎样的数据结构；其次还得有个自动解析的工具，俗话说得好，“没有金刚钻，不揽瓷器活”。对于捷径第一要素的 JSON 数据结构定义，Kotlin 特有的数据类正好派上用场，字段名、字段类型、字段默认值等色香味俱全，还有自带方法 equals、copy、toString 等下酒小菜，只要开发者轻拉珠帘便是一大桌的满汉全席。到底有多么省事，且看下面的用户信息数据类，包括姓名、年龄、身高、体重、婚否等字段的定义以及存取操作在内的完整功能，仅需一行 Kotlin 代码就全部搞定了：

```
    data class UserInfo(var name: String="", var age: Int=0, var height: Long=0L,
var weight: Float=0F, var married: Boolean=false)
```

接着解决捷径第二要素的工具使用，JSON 解析除了系统自带的 org.json 外，谷歌公司也提供了一个增强库 Gson，专门用于 JSON 串的自动解析。不过由于是第三方库，因此首先要修改模块的 build.gradle 文件，在里面的 dependencies 节点下添加下面一行配置，表示导入指定版本的 Gson 库：

```
        compile "com.google.code.gson:gson:2.8.2"
```

其次，还要在 kt 源码文件头部添加如下一行导入语句，表示后面会用到 Gson 工具类：

```
    import com.google.gson.Gson
```

完成了以上两个步骤，然后就能在代码中调用 Gson 的各种处理方法了，Gson 常用的方法有两个，一个名叫 toJson，可把数据对象转换为 JSON 字符串；另一个名叫 fromJson，可将 JSON 字符串自动解析为数据对象，该方法的代码调用格式为“fromJson(json 串, 数据类的类名::class.java)”。Kotlin 的数据类定义代码尚且只有一行，这里的 JSON 串自动解析仍旧只需一行代码，为开发者节省了不少时间。

下面是个通过 Gson 库实现 JSON 自动解析的 Kotlin 代码例子：

```
    class JsonConvertActivity : AppCompatActivity() {
        private val user = UserInfo(name="阿四", age=25, height=160L, weight=45.0f,
married=false)
        //把数据类的对象直接转换成 json 格式
        private val json = Gson().toJson(user)

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_json_convert)
            btn_origin_json.setOnClickListener { tv_json.text = "json 串内容如下：
\n$json" }
            btn_convert_json.setOnClickListener {
                //利用 Gson 包直接将 json 串解析为对应格式的数据类对象
```

```
            val newUser = Gson().fromJson(json, UserInfo::class.java)
            tv_json.text = "从 json 串解析而来的用户信息如下：" +
                    "\n\t 姓名=${newUser.name}" +
                    "\n\t 年龄=${newUser.age}" +
                    "\n\t 身高=${newUser.height}" +
                    "\n\t 体重=${newUser.weight}" +
                    "\n\t 婚否=${newUser.married}"
        }
    }
}
```

上述 JSON 串自动解析前后的效果分别如图 10-9 和图 10-10 所示，其中图 10-9 展示待解析的 JSON 字符串内容，图 10-10 展示按照数据类格式自动解析之后的各字段值。

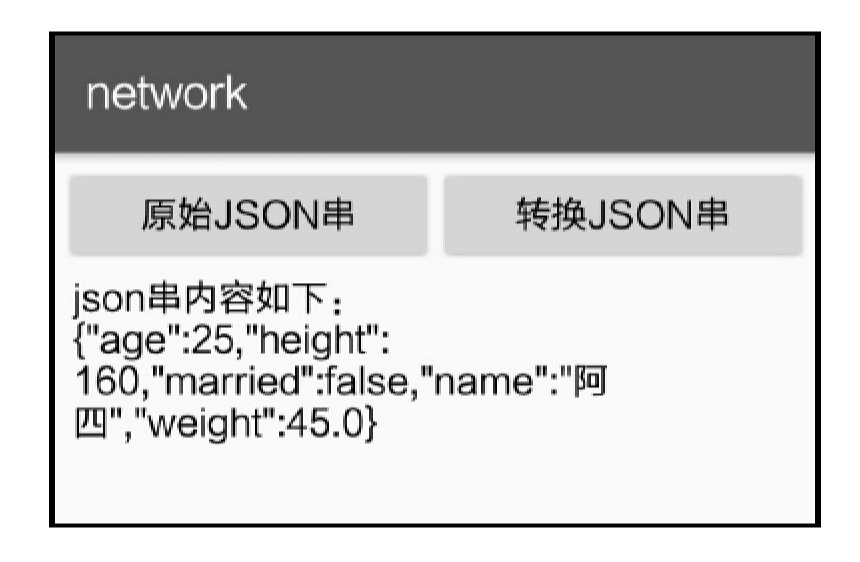


图 10-9　自动解析前的 JSON 字符串

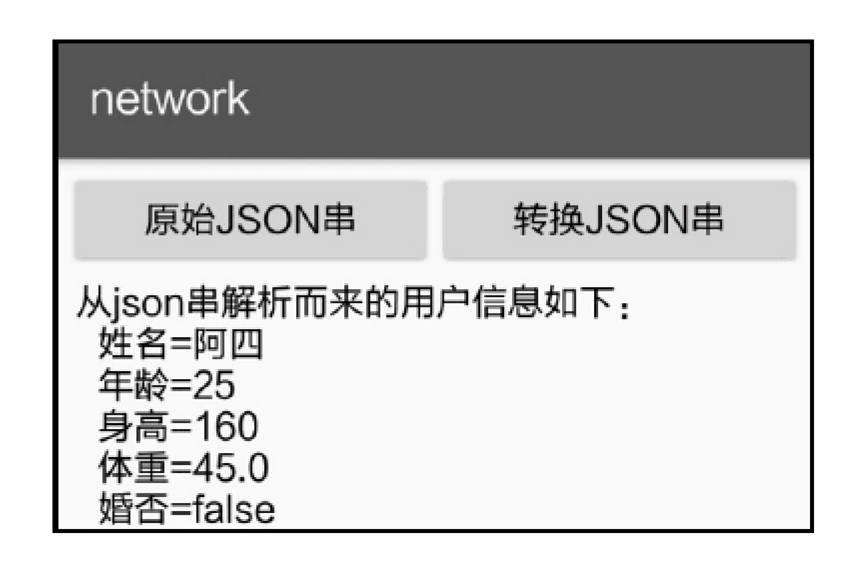


图 10-10　自动解析后的数据类字段

10.2.3　HTTP 接口调用

手机上的图文资源毕竟有限，为了获取更丰富的信息，就得到辽阔的互联网大海上冲浪。App 自身也要经常与服务器交互，以便获取最新的数据显示到界面上。这个客户端与服务端之间的信息交互功能基本使用 HTTP 协议进行通信，即 App 访问服务器的 HTTP 接口来传输数据。HTTP 接口调用在 Java 代码中可不是一个轻松的活，开发者若用最基础的 HttpURLConnection 来编码，则至少要考虑以下场景的处理：

（1）HTTP 的请求方式是什么，是 GET、POST、PUT 还是 DELETE？

（2）HTTP 的连接超时时间是多少，请求应答的超时时间又是多少？

（3）HTTP 头部的语言和浏览器信息该怎么设置？

（4）HTTP 传输的数据内容采取的是哪种编码方式？

（5）HTTP 的应答数据如果是压缩过的，又要如何解压？

（6）HTTP 的输入输出流需要注意哪些方面？

（7）HTTP 如何分块传输较大的数据信息？

瞧瞧上面层出不穷的功能要求，如果开发者事必躬亲逐个编码，那可真是要累得够呛。因此，各种意图取代 HttpURLConnection 的网络交互框架如雨后春笋般涌现出来，既有老资格的如 HttpClient，又有后起之秀如 Android-Async-Http、Volley、OkHttp、Retrofit 等，可谓是百花齐放、百家争鸣。当然，这些网络框架是需要学习成本的，使用起来也不如想象中的那么容易，它们只是

在技术上各有千秋，并非终极的解决方案，往往是你方唱罢我登台，各领风骚几年然后歇菜。

其实，HTTP 交互原本无须这样大动干戈，常见的接口调用仅仅是 App 往服务器发送一串请求信息，然后服务器返回给 App 一串处理结果，这种简单的业务场景已经足够应付大多数应用的网络通信需求。所以大道至简，Kotlin 把网络交互看作是跟文件读写一样的 I/O 操作，后端的服务地址就像是一个文件路径，于是请求服务器的数据犹如读取文件内容。同时，文本分为文本文件和二进制文件两种，则 HTTP 接口对应获取文本数据和获取二进制数据两种方式，于是整个网络请求便简化为网络数据的保存跟读取了。

具体到详细的 Kotlin 编码，既然文件对象由“File(文件路径)”构建，那么地址对象就由“URL（网络地址）”构建。获取接口数据则有 readText 和 readBytes 两种方法，前者用于获取文本形式的应答数据，后者用于获取二进制形式的应答数据，如图片文件、音频文件等。仅仅一个 readText 方法真的能完成繁杂的 HTTP 接口调用操作吗？下面通过一个具体的接口访问案例探讨一下如何使用 Kotlin 代码实现 HTTP 接口调用。

智能手机普遍提供了定位功能，可是系统自带的定位服务只能获得用户所在的经纬度信息，而这些枯燥的经纬度数字令人不知所云，肯定要把经纬度转换为详细的地址信息才方便用户理解。如果将经纬度转换为详细地址，就要访问谷歌地图提供的地址查询接口，该接口的地址形如“http://maps.google.cn/maps/api/geocode/json?请求参数信息”，App 把经纬度数据作为请求参数传入，对方会返回一个包含地址信息的 JSON 串，通过解析 JSON 串即可获得当前的详细地址。

由于访问网络需要在分线程进行，因此接口调用代码必须放在 doAsync 代码块中，下面给出根据经纬度获取详细地址的 Kotlin 代码片段：

```
    private val mapsUrl = "http://maps.google.cn/maps/api/geocode/
json?latlng={0},{1}&sensor=true&language=zh-CN"

    //位置监听器侦听到定位变化事件，就调用该函数请求详细地址
    private fun setLocationText(location: Location?) {
        if (location != null) {
            doAsync {
                //根据经纬度数据从谷歌地图获取详细地址信息
                val url = MessageFormat.format(mapsUrl, location.latitude,
location.longitude)
                val text = URL(url).readText()
                val obj = JSONObject(text)
                val resultArray = obj.getJSONArray("results")
                var address = ""
                //解析 json 字符串，其中 formatted_address 字段为具体地址名称
                if (resultArray.length() > 0) {
                    val resultObj = resultArray.getJSONObject(0)
                    address = resultObj.getString("formatted_address")
                }
                //获得该地点的详细地址之后，回到主线程把地址显示在界面上
                uiThread { findAddress(location, address) }
            }
```

```
            } else {
                tv_location.text = "$mLocation\n 暂未获取到定位对象"
            }
        }

        //在主线程中把定位信息连同地址信息都打印到界面上
        private fun findAddress(location: Location, address: String) {
            tv_location.text = "$mLocation\n 定位对象信息如下：  " +
                    "\n\t 时间：${DateUtil.nowDateTime}" +
                    "\n\t 经度：${location.longitude}，纬度：${location.latitude}" +
                    "\n\t 高度：${location.altitude}米，精度：${location.accuracy}米" +
                    "\n\t 地址：$address"
        }
```

涉及网络交互的接口请求操作需要事先声明该 App 的互联网权限。另外，上述例子也要声明定位权限，即在 AndroidManifest.xml 中添加相应的权限声明，具体的权限声明配置信息如下所示：

```
        <!-- 互联网 -->
        <uses-permission android:name="android.permission.INTERNET" />
        <!-- 定位 -->
        <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
        <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

前述的 Kotlin 代码看起来显然简明扼要，寥寥数行便搞定了完整的接口功能实现。如果使用 Java 代码实现类似功能，首先 HTTP 调用就得提供底层的接口访问代码，其次分线程请求网络又得专门写一个继承自 AsyncTask 的任务处理代码，末了 Activity 这边还得实现该任务的完成事件接口，真是兴师动众、劳民伤财。由此可见，Kotlin 的网络交互是革命性的，方式虽然简单，却足以应付大部分的网络通信需求，并且运行效果与 Java 代码几乎没有差别，例如调用地图接口查询地址信息，无论采用 Java 编码还是 Kotlin 编码，界面效果都如图 10-11 所示。

network

定位类型=gps
定位对象信息如下：
 时间：2017-11-21 23:27:53
 经度：119.28086499999999，纬度：
26.105498333333337
 高度：0.1米，精度：2.9米
 地址：中国福建省福州市鼓楼区军榕路 邮政编码: 350003

图 10-11　调用 HTTP 接口根据经纬度获取详细的地址信息

10.2.4　HTTP 图片获取

10.2.3 小节利用 readText 方法完成了文本数据的接口调用，当时提到了 readBytes 可用于获取二进制数据（如图片文件），那么获取网络图片是否也同样方便呢？下面就继续探讨如何使用 Kotlin 代码读取网络图片。

获取网络图片的基本流程同文本格式的接口访问一样，先通过 URL 类构建地址对象，然后在 doAsync 代码块中调用地址对象的 readBytes 方法获得图片的字节数组。将字节数组转换为位图对象，这在第 8 章的“8.3.3 读写图片文件”已经介绍过了，即利用 BitmapFactory 工具的 decodeByteArray

方法实现转换操作。转换好的位图当然可以在主线程中直接显示出来，也可以先保存为图片文件，等到需要的时候再去读取。第 8 章描述如何把位图保存为图片文件时，由于 Bitmap 相关类并未提供简单的图片保存方法，因此当时保存位图文件还着实费了一番功夫。现在保存网络图片反而无须如此折腾，这是因为获取网络图片得到了字节数组，字节数组保存为文件可是相当方便，只要调用 File 对象的 writeBytes 方法，短短一行就保存好图片了。

介绍完了网络图片的存取流程，最终的 Kotlin 编码一如既往的简单明了。下面是一个动态显示验证码的 Kotlin 页面代码例子：

```
    class HttpImageActivity : AppCompatActivity() {
        private val imageUrl = "http://222.77.181.14/ValidateCode.aspx?r=" //图
片验证码的测试地址

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_http_image)
            iv_image_code.setOnClickListener { getImageCode() }
            getImageCode()
        }

        //获取网络上的图片验证码
        private fun getImageCode() {
            iv_image_code.isEnabled = false
            doAsync {
                val url = "$imageUrl${DateUtil.getFormatTime()}"
                //获取指定 url 返回的字节数组
                val bytes = URL(url).readBytes()
                //把字节数组解码为位图数据
                val bitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
                //也可通过下面三行代码把字节数组写入文件，即生成一个图片文件
                val path = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).
toString() + "/"
                val file_path = "$path${DateUtil.getFormatTime()}.png"
                File(file_path).writeBytes(bytes)
                //获得验证码图片数据，回到主线程把验证码显示在界面上
                uiThread { finishGet(bitmap) }
            }
        }

        //在主线程中显示获得的验证码图片
        private fun finishGet(bitmap: Bitmap) {
            iv_image_code.setImageBitmap(bitmap)
            iv_image_code.isEnabled = true
        }
    }
```

看到了吧，即使是完整的 Activity 页面，Kotlin 也只需数十行代码而已。倘若使用 Java 完成同

样的功能，除了 HTTP 底层与 AsyncTask 的编码之外，还得补充 Bitmap 对象的图片保存代码。也就是说，Java 编程需要额外增加三个工具类的实现代码，只这一点，Kotlin 的效率就令人赞叹。而且，短小精悍的 Kotlin 代码并未造成任何功能缺失，以上面的图片验证码页面为例，使用 Java 编码和使用 Kotlin 编码最终的显示效果都如图 10-12 和图 10-13 所示。其中，图 10-12 展示刚打开页面的初始验证码，然后点击验证码图片，得到重新获取后的最新验证码，如图 10-13 所示。



图 10-12 初始页面上的图片验证码



图 10-13 点击刷新后的图片验证码

10.3 文件下载操作

除了 HTTP 接口调用这个常用功能之外，网络通信还有另一种普遍运用的形式，这便是文件下载操作。接口调用只能获取结构化的文本数据，而内容丰富、格式各异的流式数据往往需要下载之后再进行处理。待下载的文件格式小至图片文件、APK 安装包，大至音乐文件、影视文件，可以这么说，自从有了下载功能，手机上的多媒体娱乐才如此丰富多彩。本节就从 Android 自带的下载管理器 DownloadManager 入手，详细论述 App 实现下载相关功能的几种方式，以及如何利用 Kotlin 完成文件下载的编码。

10.3.1 下载管理器 DownloadManager

10.2.4 小节提到调用 URL 实例的 readBytes 方法可以获取小图片，可是这么做存在诸多限制，比如：

（1）无法断点续传，一旦中途失败，只能从头开始获取。
（2）只能转码为图片，难以转码成其他文件。
（3）不是真正意义上的下载操作，没法设置下载参数。

所以说，“10.2.4 HTTP 图片获取”小节的做法只能用于获取小图，如果要下载大图或者其他格式的大文件，就要另想办法。因为下载功能比较常用，而且业务功能相对统一，所以 Android 从 2.3（API 9）开始便提供了专门的下载工具——DownloadManager，用于统一管理下载操作。

下载管理器 DownloadManager 的对象从系统服务 Context.DOWNLOAD_SERVICE 中获取，它的具体使用过程分为三步：构建下载请求、进行下载操作、查询下载进度，详细说明如下。

1. 构建下载请求

要想使用下载功能，首先得构建一个下载请求，说明从哪里下载、下载参数是什么、下载的

文件保存到哪里等。这个下载请求便是 DownloadManager 的内部类 Request，该类的常用方法说明如下。

- 构造函数：指定从哪个网络地址下载文件。
- setAllowedNetworkTypes：指定允许进行下载的网络类型。允许下载的网络类型的取值说明见表 10-1，若同时允许多种网络类型，则可使用位运算符 “or” 把多种网络类型拼接起来。

表 10-1　下载任务允许网络类型的取值说明

DownloadManager.Request 类的允许网络类型	说明
NETWORK_WIFI	WIFI 网络
NETWORK_MOBILE	数据连接网络
NETWORK_BLUETOOTH	蓝牙网络

- setDestinationInExternalFilesDir：设置下载文件在本地的保存路径。如果指定目录已存在同名文件，系统就会将新下载的文件重命名，即在文件名末尾添加 “-1” “-2” 之类的序号。
- addRequestHeader：给 HTTP 请求添加头部参数。
- setMimeType：设置下载文件的媒体类型。一般无须设置，默认是服务器返回的媒体类型。
- setTitle：设置通知栏上的消息标题。若不设置，则默认标题是下载的文件名。
- setDescription：设置通知栏上的消息描述。若不设置，则默认显示系统估算的下载剩余时间。
- setVisibleInDownloadsUi：设置是否显示在系统的下载页面上。
- setNotificationVisibility：设置通知栏的下载任务可见类型。可见类型的取值说明见表 10-2。

表 10-2　下载任务的通知可见类型取值说明

DownloadManager.Request 类的通知可见类型	说明
VISIBILITY_HIDDEN	隐藏
VISIBILITY_VISIBLE	下载时可见（下载完成后消失）
VISIBILITY_VISIBLE_NOTIFY_COMPLETED	下载进行时与完成后都可见
VISIBILITY_VISIBLE_NOTIFY_ONLY_COMPLETION	只有下载完成后可见

2. 进行下载操作

构建完下载请求才能进行下载的相关操作。下面是 DownloadManager 的下载相关方法。

- enqueue：将下载请求加入任务队列中，排队等待下载。该方法返回本次下载任务的编号。
- remove：取消指定编号的下载任务。
- restartDownload：重新开始指定编号的下载任务。
- openDownloadedFile：打开下载完成的文件。
- getMimeTypeForDownloadedFile：获取已下载文件的媒体类型。
- query：根据查询请求获取符合条件的结果集游标。

3. 查询下载进度

虽然下载进度可在通知栏上查看，但是有时 App 自身也想了解当前的下载进度，那要调用下载管理器的 query 方法。该方法的输入参数是一个 Query 对象，返回结果集的 Cursor 游标，这里的 Cursor 用法与 SQLite 里的 Cursor 是一样的，具体可参考第 8 章的“8.2.1 数据库帮助器 SQLiteOpenHelper”。

下面是 Query 类的常用方法说明。

- setFilterById：根据编号来过滤下载任务。
- setFilterByStatus：根据状态来过滤下载任务。
- setOnlyIncludeVisibleInDownloadsUi：是否只包含在系统下载页面上的可见任务。
- orderBy：结果集按照指定字段排序。

设置完查询请求，即可调用 DownloadManager 对象的 query 方法，获得结果集的游标对象。该游标中包含下载任务的完整字段信息，其中主要下载字段的取值说明见表 10-3。

表 10-3 下载字段的取值说明

DownloadManager 类的下载字段	说明
COLUMN_LOCAL_FILENAME	下载文件的本地保存路径（已废弃）
COLUMN_LOCAL_URI	下载文件的本地保存路径（正常使用）
COLUMN_MEDIA_TYPE	下载文件的媒体类型
COLUMN_TOTAL_SIZE_BYTES	下载文件的总大小
COLUMN_BYTES_DOWNLOADED_SO_FAR	已下载文件的大小
COLUMN_STATUS	下载状态，取值说明见表 10-4

表 10-4 下载状态的取值说明

DownloadManager 类的下载状态	说明
STATUS_PENDING	挂起，即正在等待
STATUS_RUNNING	运行中
STATUS_PAUSED	暂停
STATUS_SUCCESSFUL	成功
STATUS_FAILED	失败

注意表 10-3 中的下载字段，因为 Android 7.0 之后增强了文件访问权限，造成字段 DownloadManager.COLUMN_LOCAL_FILENAME 被废弃，所以若在 7.0 及以上版本的手机访问该字段，则会触发异常 java.lang.SecurityException。此时，要想获取下载任务对应的文件路径，只能通过字段 DownloadManager.COLUMN_LOCAL_URI 来得到。

另外，系统的下载服务还提供了三种下载事件，开发者可通过监听对应的广播消息，从而进行相应的处理。这三种下载事件的处理过程说明如下。

1. 下载完成事件

在下载完成时，系统会发出名称为 DownloadManager.ACTION_DOWNLOAD_COMPLETE（其值为字符串“android.intent.action.DOWNLOAD_COMPLETE”）的广播，因此可注册一个该广播的接收器，用来判断当前任务是否已下载完毕，并进行后续的业务处理。

2. 下载进行时的通知栏点击事件

在下载过程中，一旦用户点击通知栏上的下载任务，系统就会发出名称为 DownloadManager.ACTION_NOTIFICATION_CLICKED（其值为字符串“android.intent.action.DOWNLOAD_NOTIFICATION_CLICKED”）的广播，所以可注册该广播的接收器进行相关处理，比如跳转到该任务的下载进度页面等。

3. 下载完成后的通知栏点击事件

在不同时刻点击通知栏上的下载任务会触发不同的响应事件。若在下载未完成时点击，则触发的是系统广播 DownloadManager.ACTION_NOTIFICATION_CLICKED；若在下载完成后点击，则触发的是系统 Intent.ACTION_VIEW（即浏览行为）。对于文件的浏览行为，系统会根据媒体类型自动寻找对应 App 来打开文件。因此，开发者若要控制此时的点击行为，可以调用 Request 对象的 setMimeType 方法设置媒体类型，这样 Android 就会按照这个类型打开相应的 App。

下面是利用 DownloadManager 下载 APK 安装包的 Kotlin 代码片段，此时将下载进度显示在通知栏上：

```
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_download_apk)
            Download.tv_apk_result = findViewById<TextView>(R.id.tv_apk_result)
            tv_spinner.text = apkNames[0]
            tv_spinner.setOnClickListener {
                selector("请选择要下载的安装包", apkNames) { i ->
                    tv_spinner.text = apkNames[i]
                    toast("${apkNames[i]}正在下载，详情见通知栏")
                    //声明下载任务的请求对象
                    val down = Request(Uri.parse(apkUrls[i]))
                    //指定通知栏上的标题文本
                    down.setTitle("${apkNames[i]}下载信息")
                    //指定通知栏上的描述文本
                    down.setDescription("${apkNames[i]}安装包正在下载")
                    //手机连上移动网络或者连上 WIFI 时均可进行下载操作
                    down.setAllowedNetworkTypes(Request.NETWORK_MOBILE or
Request.NETWORK_WIFI)
                    //指定下载通知栏在下载中与完成后都可见

down.setNotificationVisibility(Request.VISIBILITY_VISIBLE_NOTIFY_COMPLETED)
                    //指定显示在系统的下载页面上
                    down.setVisibleInDownloadsUi(true)
                    //指定下载文件在本地的保存路径
```

```
                down.setDestinationInExternalFilesDir(this,
                        Environment.DIRECTORY_DOWNLOADS, "$i.apk")
                //把下载请求添加到下载队列中
                //这里利用扩展属性实现了自动获取下载管理器实例
                //有关扩展属性的介绍参见第 9 章的“9.5.2 开始热身：震动器 Vibrator”
                downloadId = downloader.enqueue(down)
            }
        }
    }

    // 接收下载完成事件
    class DownloadCompleteReceiver : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            if (intent.action == DownloadManager.ACTION_DOWNLOAD_COMPLETE &&
Download.tv_apk_result != null) {
                //获取下载任务的编号
                val downId = intent.getLongExtra
(DownloadManager.EXTRA_DOWNLOAD_ID, -1)
                Download.tv_apk_result?.visibility = View.VISIBLE
                Download.tv_apk_result?.text = "${DateUtil.getFormatTime()} 编
号${downId}的下载任务已完成"
            }
        }
    }

    // 接收下载通知栏的点击事件，在下载过程中有效，下载完成后失效
    class NotificationClickReceiver : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            if (intent.action == DownloadManager.ACTION_NOTIFICATION_CLICKED &&
Download.tv_apk_result != null) {
                //获取下载任务的编号数组
                val downIds = intent.getLongArrayExtra
(DownloadManager.EXTRA_NOTIFICATION_CLICK_DOWNLOAD_IDS)
                for (downId in downIds) {
                    //只处理当前下载任务的点击事件
                    if (downId == downloadId) {
                        Download.tv_apk_result?.text =
"${DateUtil.getFormatTime()} 编号${downId}的下载进度条被点击了一下"
                    }
                }
            }
        }
    }

    companion object Download {
```

```
        //因为 DownloadCompleteReceiver 和 NotificationClickReceiver 是嵌套类
        //嵌套类只能操作类的静态属性，所以把 tv_apk_result 和 downloadId 放在伴生对象里面
        var tv_apk_result: TextView? = null
        private var downloadId: Long = 0
    }
```

注意到上述的 Kotlin 代码有接收两类下载事件，所以要在 AndroidManifest.xml 中注册对应广播的接收器，具体注册信息如下所示：

```
        <!-- 注册下载完成事件的广播接收器 -->
        <receiver android:name=
".DownloadApkActivity$DownloadCompleteReceiver">
            <intent-filter>
                <action android:name="android.intent.action.DOWNLOAD_COMPLETE" />
            </intent-filter>
        </receiver>
        <!-- 注册下载通知栏点击事件的广播接收器 -->
        <receiver android:name=
".DownloadApkActivity$NotificationClickReceiver">
            <intent-filter>
                <action android:name="android.intent.action.
DOWNLOAD_NOTIFICATION_CLICKED" />
            </intent-filter>
        </receiver>
```

APK 文件下载的通知栏效果如图 10-14 和图 10-15 所示，其中图 10-14 所示为下载进行中的通知栏界面，图 10-15 所示为下载完成后的通知栏界面。

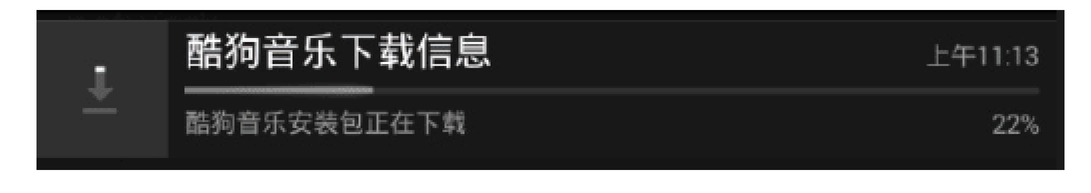


图 10-14　下载进行中的通知栏界面

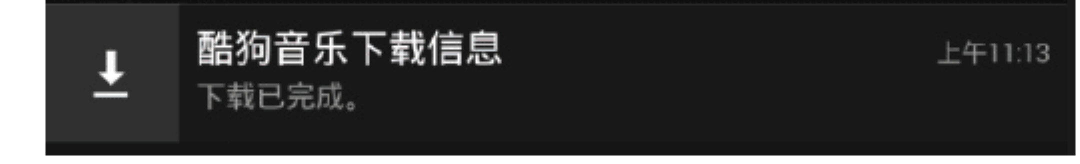


图 10-15　下载完成后的通知栏界面

10.3.2　自定义文本进度圈

10.3.1 小节把下载进度显示到通知栏上，此时进度通知由系统自动计算。但是在更多情况下，用户希望在当前界面就能看到文件的实时下载进度，而不是还要过一会儿就下拉通知栏瞅瞅。于是便要求由 App 自身去查询当前文件的下载进度，并把进度描述显示到界面的合适控件上。这里涉及两项技术细节，其一为轮询文件的下载进度，该功能参见上一小节提到的第三个步骤“3. 查询下载进度”；其二为描述进度的合适控件，该控件可考虑采用前面“10.1.2 进度对话框 ProgressDialog”小节提到的水平进度对话框。不过与水平的长条进度相比，App 使用圆圈进度更加常见，可是 ProgressBar 的圆圈样式无法设定具体的进度值，所以若要采用圆圈进度，则只好完全摒弃 ProgressBar，从头实现自定义的圆圈进度控件。

结合第 9 章的自定义视图技术，可将圆圈进度控件分解为三个绘图单元：首先绘制整个灰色

圆环作为背景；然后绘制绿色的圆弧作为前景，圆弧的角度由进度数值决定，进度越大则圆弧对应的角度也越大；最后在圆圈中央（即圆心附近）绘制进度文本，例如“50%”。按照以上三个绘图单元的划分，下面给出自定义文本进度圈的 Kotlin 实现代码例子：

```
    //自定义视图要在类名后面增加“@JvmOverloads constructor”，因为布局文件中的自定义视图必须兼容 Java
    class TextProgressCircle @JvmOverloads constructor(private val mContext: Context, attr: AttributeSet? = null) : View(mContext, attr) {
        private val paintBack: Paint = Paint()
        private val paintFore: Paint = Paint()
        private val paintText: Paint = Paint()
        private var lineWidth = 10
        private var lineColor = Color.GREEN
        private var mTextSize = 50.0f
        private lateinit var mRect: RectF
        private var mProgress = 0

        init {
            //初始化背景画笔的相关属性
            paintBack.isAntiAlias = true
            paintBack.color = Color.LTGRAY
            paintBack.strokeWidth = lineWidth.toFloat()
            paintBack.style = Style.STROKE
            //初始化前景画笔的相关属性
            paintFore.isAntiAlias = true
            paintFore.color = lineColor
            paintFore.strokeWidth = lineWidth.toFloat()
            paintFore.style = Style.STROKE
            //初始化文本画笔的相关属性
            paintText.isAntiAlias = true
            paintText.color = Color.BLUE
            paintText.textSize = mTextSize
        }

        //重写 onDraw 绘图函数，绘制圆圈背景、圆圈前景以及中央的进度文本
        override fun onDraw(canvas: Canvas) {
            super.onDraw(canvas)
            val width = measuredWidth  //获得当前视图的丈量宽度
            val height = measuredHeight  //获得当前视图的丈量高度
            if (width <= 0 || height <= 0) {
                return
            }
            val diameter = Math.min(width, height)
            mRect = RectF(((width - diameter) /2 + lineWidth).toFloat(), ((height - diameter) /2 + lineWidth).toFloat(),
```

```
                    ((width + diameter) / 2 - lineWidth).toFloat(), ((height +
diameter) / 2 - lineWidth).toFloat())
            //绘制进度圆圈的背景，背景是完整的圆环(360 度绘制)
            canvas.drawArc(mRect, 0f, 360f, false, paintBack)
            //绘制进度圆圈的前景，前景是实际进度占 360 度的百分比
            canvas.drawArc(mRect, 0f, (mProgress * 360 / 100).toFloat(), false,
paintFore)
            val text = "${mProgress.toString()}%"
            val rect = Rect()
            //获得进度文本的矩形边界
            paintText.getTextBounds(text, 0, text.length, rect)
            val x = getWidth() / 2 - rect.centerX()
            val y = getHeight() / 2 - rect.centerY()
            //把文本内容绘制在进度圆圈的圆心位置
            canvas.drawText(text, x.toFloat(), y.toFloat(), paintText)
        }

        //设置进度数值以及进度文本的文字大小
        fun setProgress(progress: Int, textSize: Float) {
            mProgress = progress
            if (textSize > 0) {
                mTextSize = textSize
                paintText.textSize = mTextSize
            }
            invalidate()
        }

        //设置进度圆圈的线宽与颜色
        fun setProgressStyle(line_width: Int, line_color: Int) {
            if (line_width > 0) {
                lineWidth = line_width
                paintFore.strokeWidth = lineWidth.toFloat()
            }
            if (line_color > 0) {
                lineColor = line_color
                paintFore.color = lineColor
            }
            invalidate()
        }
    }
```

然后在测试页面添加上述的文字进度圈控件，运行之后的显示效果如图 10-16 和图 10-17 所示，其中图 10-16 展示进度为 30%时的进度圈界面，图 10-17 展示进度为 70%时的进度圈界面。

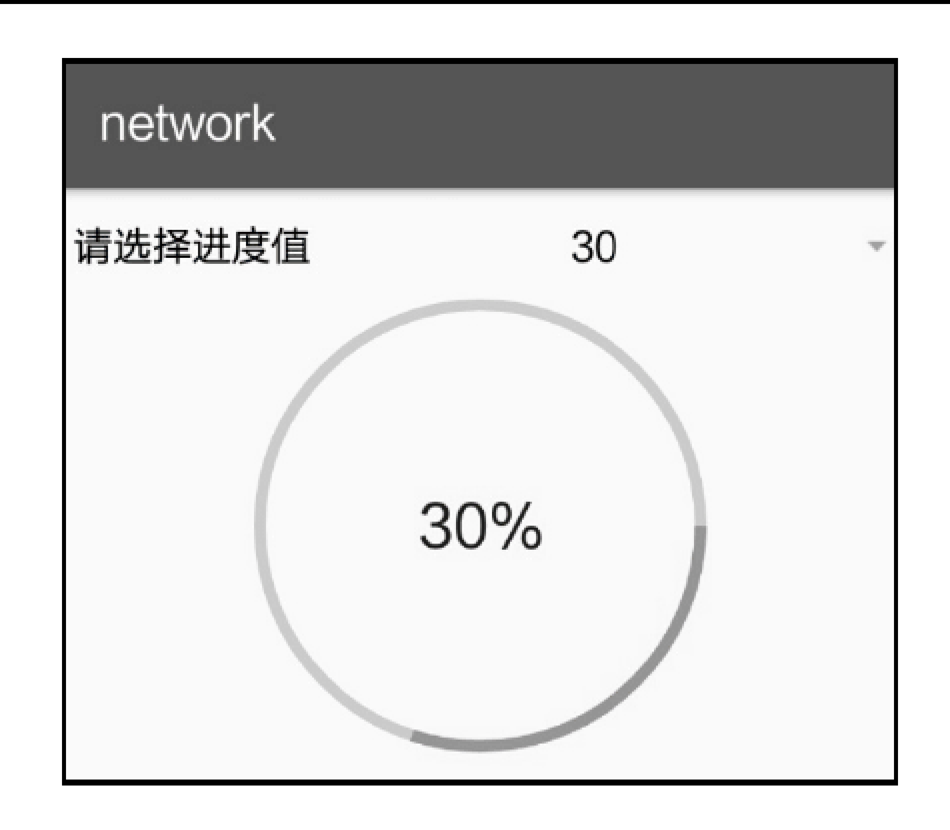



图 10-16　进度为 30%的文本圈效果

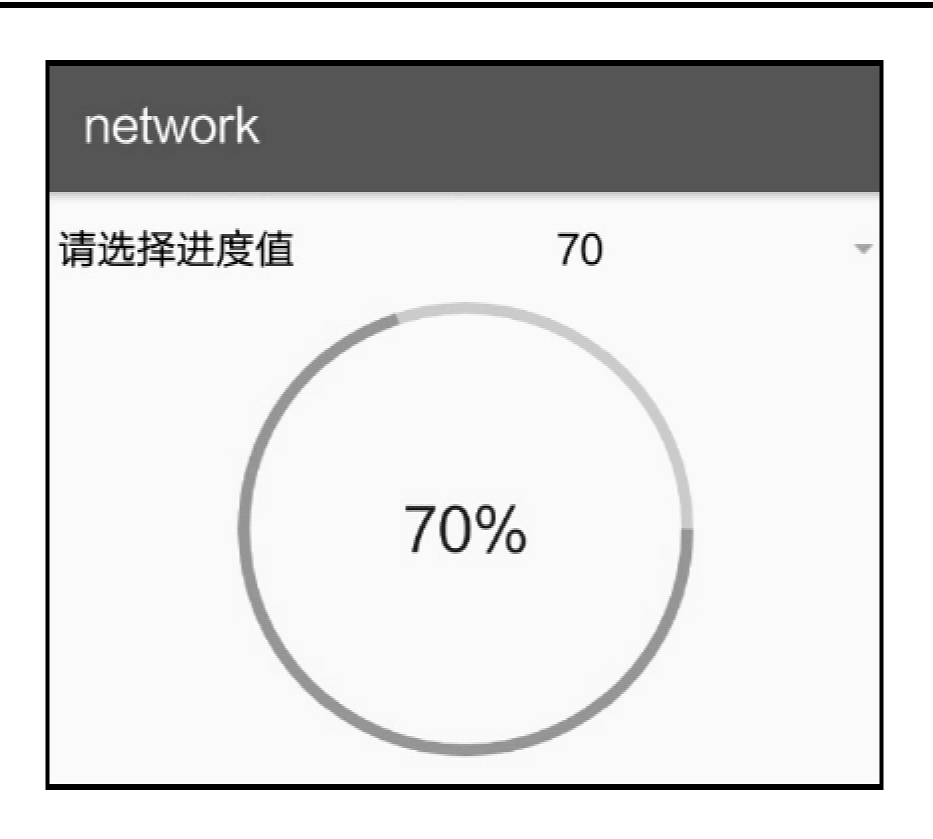


图 10-17　进度为 70%的文本圈效果

10.3.3　在页面上动态显示下载进度

现在有了 10.3.2 小节的自定义文字进度圈做铺垫，就能把轮询到的实时下载进度显示为进度圆弧。举个图片下载的例子，先在页面上预留待下载图片的位置，然后在尚未下载完成的时候，原图片位置展示包含下载进度的文字进度圈，等到下载任务完成的时候，图片位置关闭文字进度圈，改为展示已下载的图片界面。

下面是在页面上展示图片下载进度的 Kotlin 代码片段：

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_download_image)
        tv_spinner.text = imageNames[0]
        tv_spinner.setOnClickListener {
            selector("请选择要下载的图片", imageNames) { i ->
                tv_spinner.text = imageNames[i]
                tv_spinner.isEnabled = false
                iv_image_url.setImageDrawable(null)
                tpc_progress.setProgress(0, 100f)
                tpc_progress.visibility = View.VISIBLE
                //声明下载任务的请求对象
                val down = Request(Uri.parse(imageUrls[i]))
                //手机连上移动网络或者连上 WIFI 时均可进行下载操作
                down.setAllowedNetworkTypes(Request.NETWORK_MOBILE or
Request.NETWORK_WIFI)
                //隐藏下载通知栏
                down.setNotificationVisibility(Request.VISIBILITY_HIDDEN)
                //指定不在系统的下载页面显示
                down.setVisibleInDownloadsUi(false)
                //指定下载文件在本地的保存路径
                down.setDestinationInExternalFilesDir(
                        this, Environment.DIRECTORY_DCIM, "$i.jpg")
```

```
                //把下载请求添加到下载队列中
                //这里利用扩展属性实现了自动获取下载管理器实例
                //有关扩展属性的介绍参见第 9 章的“9.5.2 开始热身：震动器 Vibrator”
                downloadId = downloader.enqueue(down)
                //启动下载进度的刷新任务
                handler.postDelayed(mRefresh, 100)
            }
        }
    }

    private val handler = Handler()
    private val mRefresh = object : Runnable {
        override fun run() {
            var bFinish = false
            val down_query = Query()
            //根据编号来过滤下载任务
            down_query.setFilterById(downloadId)
            // 根据查询请求获取符合条件的结果集游标
            val cursor = downloader.query(down_query)
            while (cursor.moveToNext()) {
                //获取下载文件的 uri 路径
                val uriIdx = cursor.getColumnIndex
(DownloadManager.COLUMN_LOCAL_URI)
                //获取文件的媒体类型
                val mediaTypeIdx = cursor.getColumnIndex
(DownloadManager.COLUMN_MEDIA_TYPE)
                //获取文件的总大小
                val totalSizeIdx = cursor.getColumnIndex
(DownloadManager.COLUMN_TOTAL_SIZE_BYTES)
                //获取已下载的文件大小
                val nowSizeIdx =
cursor.getColumnIndex(DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR)
                //获取文件的下载状态
                val statusIdx = cursor.getColumnIndex
(DownloadManager.COLUMN_STATUS)
                //计算当前的下载进度百分比
                val progress = (100 * cursor.getLong(nowSizeIdx) /
cursor.getLong(totalSizeIdx)).toInt()
                if (cursor.getString(uriIdx) == null) {
                    break
                }
                //设置文本进度圈的当前进度
                tpc_progress.setProgress(progress, 100f)
                imagePath = cursor.getString(uriIdx)
                tv_image_result.text = "文件路径：${cursor.getString(uriIdx)}\n" +
```

```
                    "媒体类型：${cursor.getString(mediaTypeIdx)}\n" +
                    "文件总大小：${cursor.getLong(totalSizeIdx)}\n" +
                    "已下载大小：${cursor.getLong(nowSizeIdx)}\n" +
                    "下载进度：$progress%%\n" +
                    "下载状态：${statusMap[cursor.getInt(statusIdx)]}\n"
                //下载进度达到100%，表示下载完成
                if (progress >= 100) {
                    bFinish = true
                }
            }
            cursor.close()
            //尚未完成下载，继续轮询下载进度
            if (!bFinish) {
                handler.postDelayed(this, 100)
            } else {
                tv_spinner.isEnabled = true
                //下载完毕，隐藏圆圈进度，改为显示下载好的图片
                tpc_progress.visibility = View.INVISIBLE
                iv_image_url.setImageURI(Uri.parse(imagePath))
            }
        }
    }

    companion object {
        //下载状态类型与中文名称的映射关系定义
        private val statusMap = mapOf(
                Pair(DownloadManager.STATUS_PENDING, "挂起"),
                Pair(DownloadManager.STATUS_RUNNING, "运行中"),
                Pair(DownloadManager.STATUS_PAUSED, "暂停"),
                Pair(DownloadManager.STATUS_SUCCESSFUL, "成功"),
                Pair(DownloadManager.STATUS_FAILED, "失败"))
    }
```

上述 Kotlin 代码指定了隐藏通知栏上的下载进度，也就是将通知的可见类型设置为 VISIBILITY_HIDDEN，此时需要在 AndroidManifest.xml 中加入对应权限，具体的权限配置如下所示（含网络访问权限）：

```
    <!-- 互联网 -->
<uses-permission android:name="android.permission.INTERNET" />
    <!-- 下载时不提示通知栏 -->
    <uses-permission
android:name="android.permission.DOWNLOAD_WITHOUT_NOTIFICATION" />
```

在页面上动态展示图片下载进度的效果如图 10-18 和图 10-19 所示，进度形式采用 10.3.2 小节介绍的文字进度圈，在下载过程中显示带百分比文字的进度圆圈，下载完成后显示已下载的图片。其中，图 10-18 所示为刚开始下载的界面，此时进度是 5%，并且采用进度圆圈占位；图 10-19 所

示为下载完毕的界面，此时占位用的进度圆圈消失，取而代之的是下载到本地的图片。



图 10-18　刚开始下载图片时的界面



图 10-19　图片下载完成后的界面

10.4　ContentProvider 内容提供

ContentProvider 号称是 Android 四大组件之一（其他三个是活动 Activity、广播 Broadcast、服务 Service），主要用于在不同的 App 之间共享数据。虽然 ContentProvider 是四大组件之一，但其内部又分为三个内容组件，分别是内容提供器 ContentProvider、内容解析器 ContentResolver、内容观察器 ContentObserver。本节就对这三种内容组件进行详细的介绍。

10.4.1　内容提供器 ContentProvider

ContentProvider 为 App 存取内部数据提供了统一的外部接口，它让不同的应用之间得以共享数据。像开发者熟知的 SQLite，操作的是应用自身的内部数据库；文件的上传和下载操作的是后端服务器的外部文件；而 ContentProvider 操作本设备其他应用的内部数据，是一种中间层次的数据存储形式，即调用者位于手机内部，却位于当前 App 外部。

实际编码中，ContentProvider 只是一个服务端的数据存取接口，开发者需要在其基础上实现一个具体类，并重写以下相关的数据库管理方法。

- onCreate：创建数据库并获得数据库连接。
- query：查询数据。
- insert：插入数据。

- update：更新数据。
- delete：删除数据。
- getType：获取数据类型。

这些方法看起来是不是很像 SQLite？没错，ContentProvider 作为中间的接口，本身并不直接保存数据，而是通过 SQLiteOpenHelper 与 SQLiteDatabase 间接操作底层的 SQLite。所以要想使用 ContentProvider，首先得实现 SQLite 的数据表帮助器，再由 ContentProvider 封装对外的接口。

下面是使用 ContentProvider 提供用户信息对外接口的 Kotlin 代码例子：

```
    class UserInfoProvider : ContentProvider() {
        lateinit var userDB: UserDBHelper

        //删除数据
        override fun delete(uri: Uri, selection: String?, selectionArgs:
Array<String>?): Int {
            var count = 0
            if (uriMatcher.match(uri) == USER_INFO) {
                count = userDB.delete(selection, selectionArgs)
            }
            return count
        }

        //插入数据
        override fun insert(uri: Uri, values: ContentValues?): Uri? {
            var newUri = uri
            if (uriMatcher.match(uri) == USER_INFO) {
                // 向指定的表插入数据，得到返回的 Id
                val rowId = userDB.insert(values)
                if (rowId > 0) { // 判断插入是否执行成功
                    // 如果添加成功，就利用新添加的 Id 和生成的新地址
                    newUri = ContentUris.withAppendedId
(UserInfoContent.CONTENT_URI, rowId)
                    // 通知监听器，数据已经改变
                    context.contentResolver.notifyChange(newUri, null)
                }
            }
            return newUri
        }

        //创建 ContentProvider 时调用的回调函数
        override fun onCreate(): Boolean {
            userDB = UserDBHelper.getInstance(context, 1)
            return false
        }
```

```
        //查询数据库
        override fun query(uri: Uri, projection: Array<String>?, selection: 
String?,
                        selectionArgs: Array<String>?, sortOrder: String?): 
Cursor? {
            var cursor: Cursor? = null
            if (uriMatcher.match(uri) == USER_INFO) {
                // 执行查询
                cursor = userDB.query(projection, selection, selectionArgs, 
sortOrder)
                // 设置监听
                cursor?.setNotificationUri(context.contentResolver, uri)
            }
            return cursor

        }

        //获取数据访问类型，暂未实现
        override fun getType(uri: Uri): String? {
            throw UnsupportedOperationException("Not yet implemented")
        }

        //更新数据，暂未实现
        override fun update(uri: Uri, values: ContentValues?, selection: String?, 
selectionArgs: Array<String>?): Int {
            throw UnsupportedOperationException("Not yet implemented")
        }

        companion object {
            val USER_INFO = 1
            val uriMatcher = UriMatcher(UriMatcher.NO_MATCH)
            //伴生对象的初始化操作
            init {
                uriMatcher.addURI(UserInfoContent.AUTHORITIES, "/user", 
USER_INFO)
            }
        }
    }
```

既然内容提供器是四大组件之一，就得在 AndroidManifest.xml 中注册它的定义，并对其开放外部访问的权限，注册代码示例如下：

```
            <!-- 注册用户信息的内容提供器 -->
            <provider
                android:name=".provider.UserInfoProvider"
                android:authorities=
```

```
"com.example.network.provider.UserInfoProvider"
              android:enabled="true"
              android:exported="true" />
```

provider 注册完毕，如此便完成了服务端 App 的封装工作，接下来即可由其他 App 进行数据存取。

10.4.2 内容解析器 ContentResolver

10.4.1 小节提到了利用 ContentProvider 实现服务端 App 的数据封装，如果其他 App 想访问服务端 App 的内部数据，就要通过内容解析器 ContentResolver 来访问。内容解析器是外部 App 操作服务端数据的工具，相对应的内容提供器是服务端的数据接口。若要获取 ContentResolver 对象，则可在 Activity 代码中调用 getContentResolver 方法，现在 Kotlin 中可直接使用属性 ContentResolver 取代方法 getContentResolver()。

ContentResolver 提供的方法与 ContentProvider 是一一对应的，比如 query、insert、update、delete、getType 等方法，连方法的参数类型都一模一样。其中，最常用的是 query 方法，调用该方法返回一个游标 Cursor 对象，这个游标与 SQLite 的游标是同样的，想必读者早已用得炉火纯青。

下面是 query 方法的具体参数说明。

- uri：Uri 类型，可以理解为本次操作的数据表路径。
- projection：字符串数组类型，指定将要查询的字段名称列表。
- selection：字符串类型，指定查询条件。
- selectionArgs：字符串数组类型，指定查询条件中的参数取值列表。
- sortOrder：字符串类型，指定排序条件。

针对 10.4.1 小节 UserInfoProvider 提供的数据接口，下面是使用 ContentResolver 在外部 App 添加用户信息的 Kotlin 代码例子：

```
    private fun addUser(resolver: ContentResolver, user: UserData) {
        val name = ContentValues()
        name.put("name", user.name)
        name.put("age", user.age)
        name.put("height", user.height)
        name.put("weight", user.weight)
        name.put("married", false)
        name.put("update_time", DateUtil.getFormatTime())
        //UserInfoContent.CONTENT_URI 指向的字符串就是 provider 在
AndroidManifest.xml 里的 android:authorities 属性值
        resolver.insert(UserInfoContent.CONTENT_URI, name)
    }
```

下面是使用 ContentResolver 在客户端查询所有用户信息的代码例子：

```
    private fun readAllUser(resolver: ContentResolver): String {
        val userArray = ArrayList<UserData>()
```

```
            val cursor = resolver.query(UserInfoContent.CONTENT_URI, null, null,
null, null)
            while (cursor.moveToNext()) {
                val user = UserData()
                user.name = cursor.getString(cursor.getColumnIndex
(UserInfoContent.USER_NAME))
                user.age = cursor.getInt(cursor.getColumnIndex
(UserInfoContent.USER_AGE))
                user.height = cursor.getInt(cursor.getColumnIndex
(UserInfoContent.USER_HEIGHT)).toLong()
                user.weight = cursor.getFloat(cursor.getColumnIndex
(UserInfoContent.USER_WEIGHT))
                userArray.add(user)
            }
            cursor.close()
            var result = ""
            for (user in userArray) {
                result = "$result${user.name} 年龄${user.age} 身高${user.height}
 体重${user.weight}\n"
            }
            return result
        }
```

利用内容解析器添加用户信息的效果如图 10-20 所示，一开始服务端的用户表不存在用户记录，外部 App 通过 ContentResolver 添加一条记录后，服务端的用户记录数返回 1。用户信息的查询明细如图 10-21 所示，点击页面上的用户记录数量，弹出一个对话框，提示当前找到的所有用户的明细数据，包括姓名、年龄、身高、体重等信息。



图 10-20　添加一条用户信息的界面



图 10-21　用户信息明细的查询结果

实际开发中，普通 App 很少会开放数据接口给其他应用访问，所以作为服务端接口的 ContentProvider 反而基本用不到。内容组件能够派上用场的情况往往是 App 想要访问系统应用的通信数据，比如查看联系人、短信、通话记录以及对这些通信信息进行增删改查。

下面是使用 ContentResolver 添加联系人信息的 Kotlin 代码片段，此时访问的数据来源变成了系统自带的联系人资源库“content://com.android.contacts/”：

```
        fun addContacts(resolver: ContentResolver, contact: Contact) {
            val raw_uri = Uri.parse("content://com.android.contacts/
raw_contacts")
            val values = ContentValues()
            // 添加一条联系人的主记录，并返回唯一的联系人编号
            val contactId = ContentUris.parseId(resolver.insert(raw_uri, values))
            val uri = Uri.parse("content://com.android.contacts/data")
            // 添加联系人姓名（要根据前面获取的 id 号）
            val name = ContentValues()
            name.put("raw_contact_id", contactId)
            name.put("mimetype", "vnd.android.cursor.item/name")
            name.put("data2", contact.name)
            resolver.insert(uri, name)
            // 添加联系人的手机号码
            val phone = ContentValues()
            phone.put("raw_contact_id", contactId)
            phone.put("mimetype", "vnd.android.cursor.item/phone_v2")
            phone.put("data2", "2")
            phone.put("data1", contact.phone)
            resolver.insert(uri, phone)
            // 添加联系人的电子邮箱
            val email = ContentValues()
            email.put("raw_contact_id", contactId)
            email.put("mimetype", "vnd.android.cursor.item/email_v2")
            email.put("data2", "2")
            email.put("data1", contact.email)
            resolver.insert(uri, email)
        }
```

注意到上述代码用了 4 条 insert 语句，但业务上只添加了一个联系人信息。这样处理有个问题，就是 4 个 insert 操作不在同一个事务中，要是中间某步 insert 操作失败，那么之前插入成功的记录无法自动回滚，从而产生垃圾数据。

为了避免这种情况的发生，Android 又提供了内容操作器 ContentProviderOperation 进行批量数据的处理。内容操作器在一个请求中封装多条记录的修改动作，然后一次性提交给服务端，这就实现了在一个事务中完成多条数据的更新操作。即使某条记录处理失败，ContentProviderOperation 也能根据事务一致性原则，自动回滚本事务已经执行的修改操作。

下面是使用 ContentProviderOperation 批量添加联系人信息的 Kotlin 代码片段：

```
        fun addFullContacts(resolver: ContentResolver, contact: Contact) {
            val raw_uri = Uri.parse("content://com.android.contacts/
raw_contacts")
            val uri = Uri.parse("content://com.android.contacts/data")
            // 依次封装联系人主记录、联系人姓名、手机号码、电子邮箱的操作行为
            val op_main = ContentProviderOperation
                    .newInsert(raw_uri).withValue("account_name", null).build()
```

```
            val op_name = ContentProviderOperation
                    .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                    .withValue("mimetype", "vnd.android.cursor.item/name")
                    .withValue("data2", contact.name).build()
            val op_phone = ContentProviderOperation
                    .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                    .withValue("mimetype", "vnd.android.cursor.item/phone_v2")
                    .withValue("data2", "2").withValue("data1", contact.phone)
                    .build()
            val op_email = ContentProviderOperation
                    .newInsert(uri).withValueBackReference("raw_contact_id", 0)
                    .withValue("mimetype", "vnd.android.cursor.item/email_v2")
                    .withValue("data2", "2").withValue("data1", contact.email)
                    .build()
            // 把以上 4 个操作行为组成行为队列，并一次性处理解决该行为队列
            val operations = mutableListOf(op_main, op_name, op_phone, op_email)
            resolver.applyBatch("com.android.contacts", operations as
ArrayList<ContentProviderOperation>)
        }
```

采取批量方式添加联系人信息的效果如图 10-22 和图 10-23 所示，其中图 10-22 所示为添加之前的截图，此时联系人个数为 174 位；图 10-23 所示为添加成功之后的截图，此时联系人个数为 175 位。



图 10-22　添加联系人之前的截图



图 10-23　添加联系人之后的截图

10.4.3　内容观察器 ContentObserver

ContentResolver 获取外部数据采用的是主动查询方式，有去查就有数据，没去查就没数据。但有时 App 不但要获取以往的数据，还要实时获取新增的数据，最常见的业务场景便是短信验证码。电商 App 经常在用户注册或者付款时下发验证码短信，为了帮用户省事，App 通常会监控手机刚收到的验证码数字，并自动填入验证码输入框。这时就用到了内容观察器 ContentObserver，它给目标内容注册一个观察器，然后目标内容的数据一旦发生变化，就马上触发观察器规定好的动作，从而执行开发者预先定义的代码。

内容观察器的用法与内容提供器类似，也要从 ContentObserver 派生出一个观察器类，然后通过 ContentResolver 对象调用相应的方法注册或注销观察器。ContentResolver 与观察器有关的方法说明如下。

- registerContentObserver：注册内容观察器。
- unregisterContentObserver：注销内容观察器。
- notifyChange：通知内容观察器发生了数据变化。

为了让读者能够更好地理解，还是举个实际应用的例子。很多手机安全 App 都具备流量校准的功能，只要把特定格式的短信发送给移动运营商，就会收到运营商下发的流量校准短信，通过解析这个流量短信，即可获取详细的用户流量数据，包括月流量额度、已使用流量、未使用流量等信息。

以中国移动的手机号码为例，发送短信内容“18”到客服号码 10086，不一会儿便会收到 10086 发来的流量结果短信。下面是利用 ContentObserver 实现流量校准功能的 Kotlin 页面代码：

```
class ContentObserverActivity : AppCompatActivity() {
    private var mObserver: SmsGetObserver? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_content_observer)
        Observer.tv_check_flow = findViewById<TextView>(R.id.tv_check_flow)
        tv_check_flow.setOnClickListener {
            alert(mCheckResult, "收到流量校准短信") {
                positiveButton("确定") {}
            }.show()
        }
        btn_check_flow.setOnClickListener {
            var dialog = indeterminateProgressDialog("正在进行流量校准", "请稍候")
            dialog.show()
            //查询数据流量，移动号码的查询方式为发送短信内容“18”给“10086”
            //电信和联通号码的短信查询方式请咨询当地运营商客服热线
            sendSmsAuto("10086", "18")
            Handler().postDelayed({
                if (dialog.isShowing == true) {
                    dialog.dismiss()
                }}, 5000)
        }
        mSmsUri = Uri.parse("content://sms")
        mSmsColumn = arrayOf("address", "body", "date")
        mObserver = SmsGetObserver(this, Handler())
        //注册短信接收的内容观察器
        contentResolver.registerContentObserver(mSmsUri!!, true, mObserver!!)
    }

    override fun onDestroy() {
        //注销短信接收的内容观察器
```

```
            contentResolver.unregisterContentObserver(mObserver!!)
            super.onDestroy()
        }

        //短信发送广播，如需处理可注册该事件的 BroadcastReceiver
        private val SENT_SMS_ACTION = "com.example.network.SENT_SMS_ACTION"
        //短信接收广播，如需处理可注册该事件的 BroadcastReceiver
        private val DELIVERED_SMS_ACTION = "com.example.network.
DELIVERED_SMS_ACTION"

        fun sendSmsAuto(phoneNumber: String, message: String) {
            //声明短信发送的广播意图
            val sentIntent = Intent(SENT_SMS_ACTION)
            sentIntent.putExtra("phone", phoneNumber)
            sentIntent.putExtra("message", message)
            val sentPI = PendingIntent.getBroadcast(this, 0, sentIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //声明短信接收的广播意图
            val deliverIntent = Intent(DELIVERED_SMS_ACTION)
            deliverIntent.putExtra("phone", phoneNumber)
            deliverIntent.putExtra("message", message)
            val deliverPI = PendingIntent.getBroadcast(this, 1, deliverIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
            //要确保打开发送短信的完全权限，不是那种还需提示的不完整权限
            val smsManager = SmsManager.getDefault()
            smsManager.sendTextMessage(phoneNumber, null, message, sentPI,
deliverPI)
        }

        //定义一个短信观察器的嵌套类
        private class SmsGetObserver(private val mContext: Context, handler:
Handler) : ContentObserver(handler) {

            override fun onChange(selfChange: Boolean) {
                var sender = ""
                var content = ""
                //查询收件箱中来自 10086 的最近短信
                val selection = "address='10086' and
date>${System.currentTimeMillis()-1000*60*60}"
                val cursor = mContext.contentResolver.query(
                        mSmsUri!!, mSmsColumn, selection, null, " date desc")
                while (cursor.moveToNext()) {
                    sender = cursor.getString(0)
                    content = cursor.getString(1)
                    break
                }
                cursor.close()
```

```
                mCheckResult = "发送号码：$sender\n 短信内容：$content"
                //将解析后的短信内容显示到界面上
                Observer.tv_check_flow!!.text = "流量校准结果如下：\n\t" +
                        "总流量为：${findFlow(content, "总流量为", "MB")}\n\t" +
                        "已使用：${findFlow(content, "已使用", "MB")}\n\t" +
                        "剩余：${findFlow(content, "剩余", "MB")}"
                super.onChange(selfChange)
            }
        }

        companion object Observer {
            private var tv_check_flow: TextView? = null
            private var mCheckResult: String = ""
            private var mSmsUri: Uri? = null
            private var mSmsColumn: Array<String>? = null
            //在伴生对象中定义解析短信内容的方法
            private fun findFlow(sms: String, begin: String, end: String): String {
                val begin_pos = sms.indexOf(begin)
                if (begin_pos < 0) {
                    return "未获取"
                }
                val sub_sms = sms.substring(begin_pos)
                val end_pos = sub_sms.indexOf(end)
                return if (end_pos < 0) {
                    "未获取"
                } else sub_sms.substring(begin.length, end_pos + end.length)
            }
        }
    }
```

根据运营商短信进行流量校准的效果如图 10-24 和图 10-25 所示，其中图 10-24 所示为用户实际收到的短信内容，图 10-25 所示为 App 监视短信并解析完成的流量数据页面。

22:47
您好！您上月结转至本月的国内流量为600MB。截止2017年11月21日22时47分，您办理的套餐内含移动数据总流量为1GB176MB，已使用388MB，剩余812MB（其中您已使用388MB，副号1（159[redacted]）已使用0MB）。其中国内流量剩余812MB（含国内结转流量212MB）。免费赠送“任我看”卡1张，一天一元500m，不用不花钱，还送首月月租：http://dx.10086.cn/rzkwap【中国移动】

图 10-24　用户收到的短信内容

network
进行流量校准
流量校准结果如下：
总流量为：1GB176MB
已使用：388MB
剩余：812MB

图 10-25　App 解析后的流量数据

10.5　实战项目：电商 App 的自动升级

都说酒香不怕巷子深，又说养在深闺人未识，这两句话看似矛盾，其实指的是同一个含义，即美好的事物要想办法让大众了解才会闻名。无论是香飘十里，还是抛头露面，目的都是开展营销，也就是俗话说的做广告。开发电商 App 也是如此，要想让大家知道这里的商品质量过硬、价格公道，势必经常举办各种促销活动，比如一年一度的 618、双十一等。既然频繁搞活动，就得进行技术支撑，隔三岔五对 App 升级一下，增加几个新功能，修复一些老 BUG 等。这个 App 升级功能便是本章末尾的实战项目“电商 App 的自动升级”，这个自动升级到底用到了哪些技术？下面就来看看该项目的分析与设计。

10.5.1　需求描述

要说安装 App，可能有的小伙伴觉得那还不简单，直接找个应用商店搜索 App 名称不就行了？但初次安装与安装后升级不是一个概念，如果升级也靠用户手动到应用商店搜索并安装，那就“图样图森破”了。贴心的做法是在 App 内部提供在线更新的功能，这个在线更新又分为两种形式，一种是由用户手工选择应用菜单上的“检查更新”，另一种是 App 启动后自行判断服务器上是否有更高版本的安装包。“检查更新”的菜单项位置在每个 App 内部都不一样，App 自动进行升级判断则后台服务并没有对应的界面，所以在线更新的效果图暂且按照图 10-26 所示的样子，效果图上有两个按钮，其中“不请求接口直接弹窗”按钮模拟了用户点击“检查更新”菜单的动作，“请求服务端接口再弹窗”按钮模拟的则是 App 自动检查升级的功能。

无论是手动更新还是自动更新，结果都要调用后端服务器的版本更新接口，根据服务器的应答报文判断是否需要升级。如果服务器返回有更新的安装包，App 界面就弹窗提示用户要不要在线升级，提示窗如图 10-27 所示。注意这里有种特殊的情况，倘若 App 扫描设备发现存储卡上已经存在新版本的 APK 安装包，则提示用户本地已经有了新包，无须耗费流量即可进行升级，此时的提示窗如图 10-28 所示。

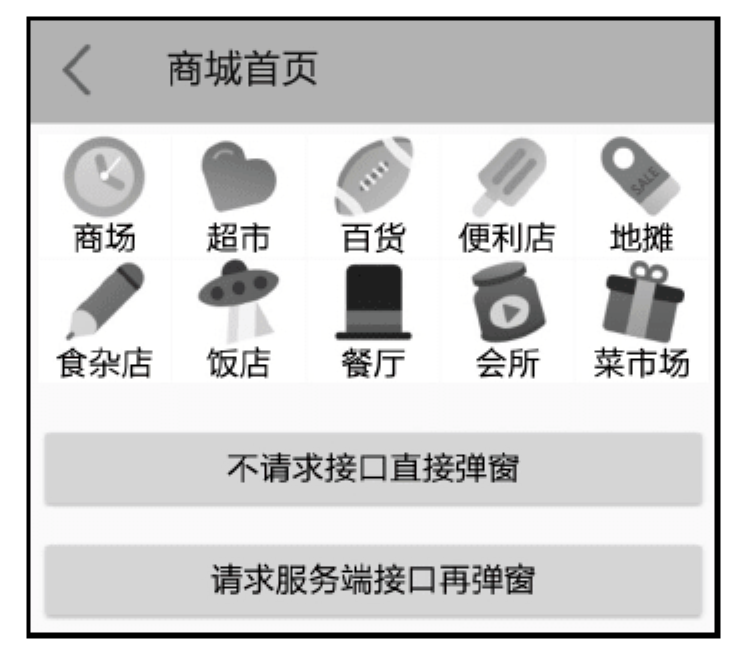


图 10-26　商城首页模拟两种升级动作

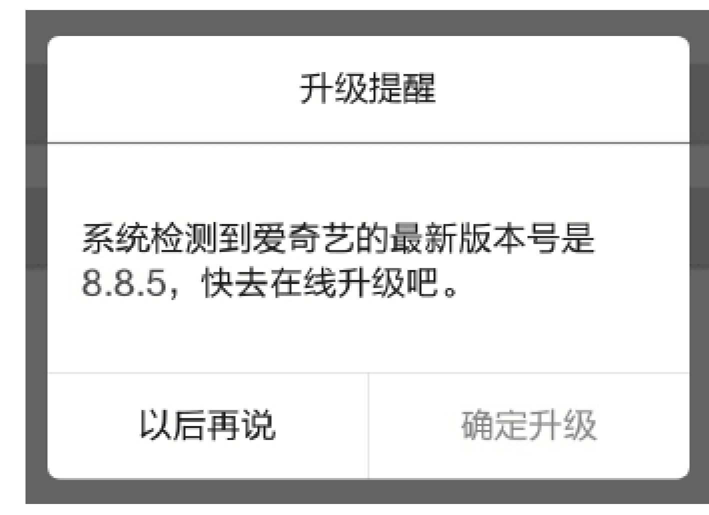


图 10-27　需要下载安装包的提示窗



图 10-28　无须下载安装包的提示窗

对于手机存储卡未找到新版本 APK 的情况，只要用户在提示框中选择“确定升级”，App 就

要一边下载 APK 文件，一边弹出进度对话框，包含下载进度的对话框如图 10-29 所示。APK 安装包下载完毕，或者 App 发现手机原来已有对应版本的安装包，接着用户确认升级操作之后，App 应当继续 APK 的安装动作，安装期间仍需弹窗提示正在安装，安装提示对话框如图 10-30 所示。

等待 App 完成新版本的升级，回到升级前的 App 界面，然后弹出升级完毕的提示对话框，如图 10-31 所示，至此方才结束 App 的自动升级历程。

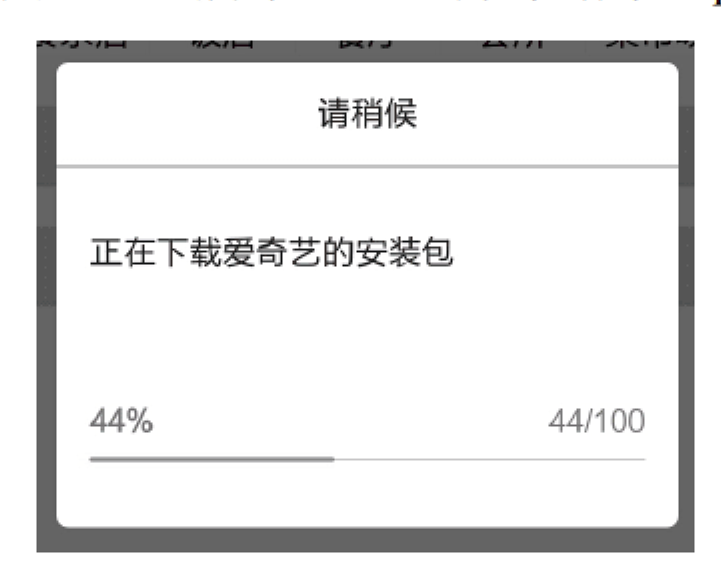


图 10-29 下载过程中的进度对话框

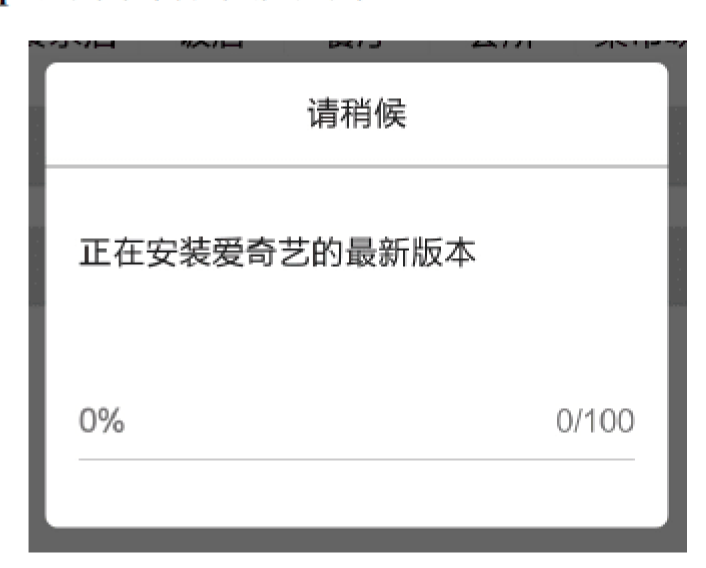


图 10-30 安装过程中的进度对话框



图 10-31 升级完毕的提示对话框

另外，上面几个提示对话框中的消息文本需要高亮显示升级后的最新版本号，从而更好地提醒用户正在升级的是哪个版本。

10.5.2 开始热身：可变字符串 SpannableString

10.5.1 小节的需求描述说到提示框中的最新版本要用高亮文字显示，可是文本视图只提供了统一的文本样式设置方法，比如通过 setTextColor 方法设置文本颜色，通过 setTextSize 方法设置文本大小，还可以通过 setTextAppearance 方法设置文本风格（包括颜色、大小、对齐方式等）。一旦调用了这些方法，文本内容就会显示同样的颜色、同样的大小或者同样的风格，根本没法让内部某些文字单独高亮显示。因此，为了解决分段文本展示不同样式的需求，Android 提供了可变字符串 SpannableString，通过该工具实现对文本样式的分段显示。

可变字符串的原理是给指定位置的文本赋予对应的样式，从而告知系统这段文本的显示方式。具体到 Kotlin 编码上，主要有三个步骤，说明如下。

步骤 01 从指定文本字符串构造一个 SpannableString 对象。

步骤 02 调用 SpannableString 对象的 setSpan 方法设置指定文本段的显示风格。该方法的第一个参数为风格样式，第二个参数为文本段的起始位置，第三个参数为文本段的终止位置，第四个参数为风格的范围标志（一般设置为 Spanned.SPAN_EXCLUSIVE_EXCLUSIVE）。

步骤 03 把处理好的 SpannableString 对象赋值给文本视图的 text 属性。注意 text 字段的类型并非 String，而是 CharSequence，因此凡是实现了 CharSequence 接口的类，其对象都允许赋值给 text 字段。字符串 String 类当然有实现 CharSequence，同样可变字符串 SpannableString 类也实现了该接口，故而它们的实例均为合法的 text 参数。

由此可见，运用可变字符串的关键是第二步，不过第二步的 setSpan 方法每次也只能单独设置一处的文字样式，倘若原字符串有多处文本需要定制文字样式，那还得多次调用 setSpan 方法，这样盘算依然不太经济。好在强大的 Anko 库又封装了一个扩展函数 buildSpanned，只要在该函数内

部调用形如“append(待定制样式的文本, 定制后的风格样式)”的代码，即可返回自动拼接后的多彩文本串。

下面是通过 buildSpanned 函数连续构建可变字符串的 Kotlin 代码例子：

```
val str: Spanned = buildSpanned {
    append("为", StyleSpan(Typeface.BOLD)) //文字字体使用粗体
    append("人民", RelativeSizeSpan(1.5f)) //文字大小增大到 1.5 倍大
    append("服务", ForegroundColorSpan(Color.RED)) //文字颜色使用红色
    append("是谁", BackgroundColorSpan(Color.GREEN)) //背景色使用绿色
    append("提出来的", UnderlineSpan()) //文字下方增加下划线
}
```

上面的 Kotlin 眼瞅着还算整齐，然而仅仅为了表示粗体就得书写完整的风格声明代码“StyleSpan(Typeface.BOLD)”着实不够干脆。所以 Anko 索性在这里快刀斩乱麻，又把“StyleSpan(Typeface.BOLD)”简化成了“Bold”，其他几个风格声明也陆续予以缩写。于是最终简化后的 Kotlin 代码如下所示：

```
val str: Spanned = buildSpanned {
    append("为", Bold) //文字字体使用粗体
    append("人民", RelativeSizeSpan(1.5f)) //文字大小增大到 1.5 倍大
    append("服务", foregroundColor(Color.RED)) //文字颜色使用红色
    append("是谁", backgroundColor(Color.GREEN)) //背景色使用绿色
    append("提出来的", Underline) //文字下方增加下划线
}
```

以上构建可变字符串用到的风格样式只是沧海一粟，完整的风格样式定义在 android.text.style 包中，总共有 30 多个类型，当然常用的没这么多，笔者整理了几个常用的风格样式，结合 Anko 库的简化写法，具体说明见表 10-5。

表 10-5 常用的显示风格列表与 Anko 库的简化写法

可变字符串的显示风格类	Anko 库的简化写法	说明
RelativeSizeSpan	无	设置文字的相对大小
StyleSpan(Typeface.BOLD)	Bold	设置粗体文字
StyleSpan(Typeface.Italic)	Italic	设置斜体文字
ForegroundColorSpan	foregroundColor	设置文字的颜色
BackgroundColorSpan	backgroundColor	设置文字的背景色
UnderlineSpan	Underline	给文字加上下划线
StrikethroughSpan	Strikethrough	给文字加上删除线
ImageSpan	无	把文字替换为图片

接着对可变字符串分别运用不同的文字样式，看看到底都有哪些风格的显示效果。下面是使用可变字符串设置文字样式的 Kotlin 代码例子：

```
    class SpannableActivity : AppCompatActivity() {
        private val spannables = listOf("增大字号", "加粗字体", "前景红色",
"背景绿色", "下划线", "表情图片", "Anko 自定义")
        private val text = "为人民服务"
        private val key = "人民"
        private var beginPos = text.indexOf(key)
        private var endPos = beginPos + key.length

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_spannable)
            tv_spannable.text = text
            tv_spinner.text = spannables[0]
            tv_spinner.setOnClickListener {
                selector("请选择可变字符串样式", spannables) { i ->
                    tv_spinner.text = spannables[i]
                    val spanText = SpannableString(text)
                    //对这段文本运用指定的风格样式
                    spanText.setSpan(when (i) {
                        0 -> RelativeSizeSpan(1.5f) //文字大小增大到 1.5 倍大
                        1 -> StyleSpan(Typeface.BOLD) //文字字体使用粗体
                        2 -> ForegroundColorSpan(Color.RED) //文字颜色使用红色
                        3 -> BackgroundColorSpan(Color.GREEN) //背景色使用绿色
                        4 -> UnderlineSpan() //文字下方增加下划线
                        else -> ImageSpan(this@SpannableActivity, R.drawable.people)
//把文字替换为图片
                        }, beginPos, endPos, Spanned.SPAN_EXCLUSIVE_EXCLUSIVE)
                    tv_spannable.text = spanText
                    if (i >= 6) {
                        //使用 Anko 库的 buildSpanned 函数连续构建可变字符串
                        tv_spannable.text = buildSpanned {
                            append("为", Bold)
                            append("人民", RelativeSizeSpan(1.5f))
                            append("服务", foregroundColor(Color.RED))
                            append("是谁", backgroundColor(Color.GREEN))
                            append("提出来的", Underline)
                        }
                    }
                }
            }
        }
    }
```

由于上面的 Kotlin 代码用到了 Anko 库的扩展函数（包括 buildSpanned、append 等），所以务必在代码头部加入下面一行导入语句：

```
import org.jetbrains.anko.*
```

另外，要修改模块的 build.gradle，在 dependencies 节点中补充下述的 anko-common 包编译配置：

```
compile "org.jetbrains.anko:anko-common:$anko_version"
```

可变字符串所呈现的不同风格效果如图 10-32～图 10-38 所示，其中图 10-32 所示为加大字体后的效果，图 10-33 所示为加粗字体后的效果，图 10-34 所示为修改文字颜色后的效果，图 10-35 所示为修改文字背景后的效果，图 10-36 所示为增加下画线后的效果，图 10-37 所示为把文字替换成图片后的效果，图 10-38 所示为采用 Anko 函数混合多种样式后的效果。



图 10-32　加大字体的字符串效果



图 10-33　加粗字体的字符串效果



图 10-34　变更文字颜色的字符串效果



图 10-35　变更背景颜色的字符串效果



图 10-36　添加下划线的字符串效果



图 10-37　文字换图片的字符串效果

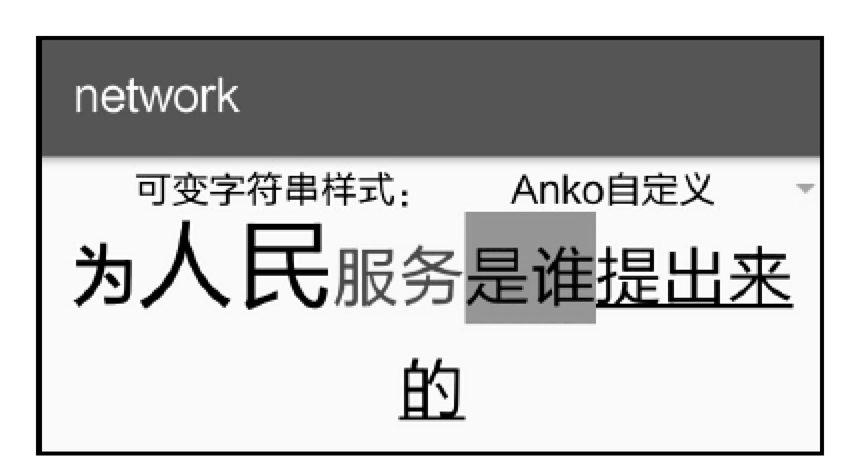


图 10-38　采用 Anko 函数混合多种样式的字符串效果

10.5.3　控件设计

由于自动升级功能更多是在后台完成的，界面上用到的控件反而不多，因此下面罗列的并不限于控件，还包括后台的处理技术。

（1）提醒对话框 AlertDialog：用于提醒用户是否立即升级应用，以及升级完毕之后的提示语。

（2）进度对话框 ProgressDialog：在下载 APK 文件的时候，以及 APK 安装过程中，都要显示进度对话框。

（3）可变字符串 SpannableString：提示文本中需要将最新版本号高亮显示，这便用到了可变字符串。

（4）多线程：App 与后端服务器进行接口交互，需要开启分线程才能调用 HTTP 接口。

（5）网络地址 URL：Kotlin 对 URL 类进行了扩展，增加了 readText 方法用于获取接口数据。

（6）移动数据格式 JSON：后端接口返回 JSON 格式的版本升级信息字符串，然后 App 把 JSON 串转换为数据类对象。

（7）下载管理器 DownloadManager：用于 APK 文件的下载行为，包括下载进度的查询操作。

（8）应用包管理器 PackageManager：根据应用包管理器获得应用的版本号信息。

除了上面提到的技术以外，也会用到内容解析器 ContentResolver，因为需求提到：如果手机内存能找到最新版本的安装包，那就无须下载直接安装即可。若要查找手机上的 APK 安装包，可到媒体资源库中查询媒体类型为“application/vnd.android.package-archive”的文件，该类型表示安卓应用的安装包，也就是 APK 文件。

10.5.4 关键代码

为了方便读者更好、更快地使用 Kotlin 编码完成自动升级项目，下面列举几个重要功能的 Kotlin 代码片段：

1. 关于向服务器请求版本更新信息

访问后端的 HTTP 接口，倘若使用 Java 编码，必定又是长篇大论。采用 Kotlin 编码的瘦身效果立竿见影，只要通过 doAsync+uiThread 组合，接口调用的操作就变得轻描淡写。下面是 HTTP 接口访问的 Kotlin 代码例子：

```
    btn_need_request.setOnClickListener {
        val pi = packageManager.getPackageInfo(packageName, 0)
        //开启分线程执行后端接口调用
        doAsync {
            //从服务端获取版本升级信息
            val url =
"$checkUrl?package_name=${pi.packageName}&version_name=${pi.versionName}"
            val result = URL(url).readText()
            //回到主线程在界面上弹窗提示待升级的版本
            uiThread { checkUpdate(result) }
        }
    }
```

既然是访问服务器的接口，肯定要有对应的服务端程序，这个服务端程序可见本书源代码中的 HttpTest 工程包。

2. 关于高亮显示一段文本中的指定文字

在已有的文本中高亮显示其中的某些文字，处理起来还有点烦琐，首先要找到指定文字在整段文本中的位置，再对这些文字设置对应的高亮样式。鉴于高亮功能比较通用，因此可以写到工具类里面，不过既然使用 Kotlin 编码，建议考虑采取扩展函数的形式将高亮处理函数作为 String 类的一个扩展函数，这样用起来更加方便。

下面是对 String 类进行扩展、添加 highlight 高亮函数的 Kotlin 代码：

```
    //字符串中的关键语句用指定样式高亮显示
    fun String.highlight(key: String, style: CharacterStyle): SpannableString {
        val spanText = SpannableString(this)
        val beginPos = this.indexOf(key)
        val endPos = beginPos + key.length
        spanText.setSpan(style, beginPos, endPos,
Spanned.SPAN_EXCLUSIVE_EXCLUSIVE)
        return spanText
    }
```

有了上述的扩展函数定义，外部就能直接调用字符串对象的 highlight 方法，完全无须记忆工具类的名称。具体的 Kotlin 调用代码如下所示：

```
    val spanText = message.highlight(vc.version_name,
ForegroundColorSpan(Color.RED))
```

同时不要忘了在代码文件头部添加下面的一行导入语句，表示此处用到了自己扩展的 highlight 方法：

```
    import com.example.network.util.highlight
```

3. 关于 alert 如何显示可变字符串的文本内容

前面刚设置好可变字符串的文本内容，不料发现无法利用 alert 方法显示该文本了，怎么回事？这是因为在 Java 编码中，AlertDialog.Builder 的 setTitle 和 setMessage 两个方法的输入参数都是 CharSequence，自然允许将可变字符串对象赋值进去。然而 Anko 库扩展出来的 alert 方法，message 和 title 这两个入参类型却改成 String 字符串了，使得 SpannableString 与 String 类型不同，因此无法输入。

没想到还有这样的事情，真叫“攻城狮”颜面何在。不过作为程序员可得不畏艰难险阻，逢山开路、遇水搭桥，现在 Anko 库造的 alert 桥过不了，不妨自己搭个自定义的新桥，偷梁换柱，把 message 字段的参数类型改成 CharSequence 就行了。于是改写后的 alert 方法代码就变成下面这样了：

```
    //Anko 自带的 alert 只支持 String 类型的文本，不支持富文本的 CharSequence 类型
    //故此处重写 alert 方法，使之支持可变字符串 SpannableString
    fun Context.alert(
            message: CharSequence,
            title: String? = null,
            init: (AlertDialogBuilder.() -> Unit)? = null
    ) = AlertDialogBuilder(this).apply {
        if (title != null) title(title)
```

```
        message(message)
        if (init != null) init()
    }
```

改写完毕，不要忘了在 Activity 代码头部添加下面的一行导入语句，表示本页面用的是自定义的 alert 方法：

```
    import com.example.network.util.alert
```

4. 关于如何获取手机上的 APK 文件

前面“10.5.3 控件设计”小节提到，利用内容解析器 ContentResolver 可到资源库中查询媒体类型为“application/vnd.android.package-archive”的 APK 文件，但这只是大概的思路。因为即便找到了几个 APK 文件，App 又如何甄别这些 APK 都是什么来头、哪个 APK 文件才符合当前应用的指定版本号呢？所以要想逐个判定 APK 文件的真实身份，还得解析 APK 内部的包信息，具体的工作则是调用 PackageManager 对象的 getPackageArchiveInfo 方法，该方法可从指定的 APK 路径获取安装包的详细数据，包括应用的包名、应用的版本号等。有了包名、版本号这些信息，App 方能鉴定本地是否存在最新版本的升级包。

下面是获取并校验本设备上 APK 文件的 Kotlin 代码例子：

```
        private fun getLocalPath(vc: VersionCheck): String {
            var local_path = ""
            //遍历本地所有的 apk 文件
            val cursor = contentResolver.query(MediaStore.Files.getContentUri
("external"),
                    null, "mime_type=\"application/vnd.android.package-archive\"",
null, null)
            while (cursor.moveToNext()) {
                //TITLE 获取文件名，DATA 获取文件完整路径，SIZE 获取文件大小
                val path = cursor.getString(cursor.getColumnIndex
(MediaStore.Files.FileColumns.DATA))
                //从 apk 文件中解析得到该安装包的程序信息
                val pi = packageManager.getPackageArchiveInfo(path,
PackageManager.GET_ACTIVITIES)
                if (pi != null) {
                    //找到包名与版本号都符合条件的 apk 文件
                    if (vc.package_name==pi.packageName &&
vc.version_name==pi.versionName) {
                        local_path = path
                    }
                }
            }
            cursor.close()
            return local_path
        }
```

5. 关于下载 APK 文件的操作过程

如果手机上没找到符合条件的安装包，就必须联网去服务器下载最新的 APK 文件。相关文件下载的 Kotlin 处理代码如下所示：

```
    //开始执行升级处理。如果本地已有安装包，就直接进行操作；如果不存在，就从网络下载安装包
    private fun startInstallApp(vc: VersionCheck) {
        appVc = vc
        //本地路径非空，表示存储卡找到最新版本的安装包，此时无须下载即可进行安装操作
        if (vc.local_path.isNotEmpty()) {
            handler.postDelayed(mInstall, 100)
        } else {
            //构建安装包下载地址的请求任务
            val down = Request(Uri.parse(vc.download_url))
            down.setAllowedNetworkTypes(Request.NETWORK_MOBILE or
Request.NETWORK_WIFI)
            //隐藏通知栏上的下载消息
            down.setNotificationVisibility(Request.VISIBILITY_HIDDEN)
            down.setVisibleInDownloadsUi(false)
            //指定下载文件的保存路径
            down.setDestinationInExternalFilesDir(this,
                    Environment.DIRECTORY_DOWNLOADS, "${vc.package_name}.apk")
            //将请求任务添加到下载队列中
            downloadId = downloader.enqueue(down)
            handler.postDelayed(mRefresh, 100)
            //弹出进度对话框，用于展示下载进度
            val message = "正在下载${appVc.app_name}的安装包"
            dialog = progressDialog(message, "请稍候")
            dialog.show()
        }
    }
```

10.6　小　　结

本章主要介绍了 Kotlin 如何进行网络通信的编程实现，包括多线程相关技术的运用（线程与消息机制、进度对话框的两种形式、异步任务的新型写法）、HTTP 接口的访问操作（JSON 串的手工解析与自动解析、HTTP 接口调用的简单实现、获取 HTTP 图片的简单实现）、文件下载的相关处理（下载管理器的用法、自定义文本进度圈、在页面上动态显示下载进度）以及 Content 内容组件的数据存储和读取（内容提供器、内容解析器、内容观察器）。最后设计了一个实战项目“电商 App 的自动升级”，在该项目的 Kotlin 编码中采用了前面介绍的部分网络通信技术，以及通过内容组件自动判断是否存在已下载的安装包，另外还介绍了 Kotlin 对可变字符串的改进编码。

通过本章的学习，读者应能掌握以下 5 种开发技能：

（1）学会使用Kotlin实现多线程任务的开发，重点掌握Kotlin对线程、进度对话框、异步任务的简要写法。

（2）学会使用Kotlin完成HTTP接口的访问编码，重点掌握Kotlin如何自动解析JSON串、如何便捷调用HTTP接口、如何快速获取HTTP图片。

（3）学会使用 Kotlin 进行文件下载的相关操作，重点掌握 Kotlin 对下载事件的处理，以及Kotlin是怎样轮询下载进度的。

（4）学会使用Kotlin开发Content内容组件的功能，例如封装数据的对外接口，以及对开放内容接口的系统数据进行查询、修改和监视操作等。

（5）学会利用Kotlin简化可变字符串的编码过程。